Pomona College
Department of Computer Science

# Rust vs. D: Exploring Possible Successors of C++

Andy Russell

May 4, 2015

Submitted as part of the senior exercise for the degree of
Bachelor of Arts in Computer Science

Professor Kim Bruce

## Abstract

The programming languages D and Rust aim to simplify the complex and error-prone features of C++ while maintaining a similar level of performance. This paper examines whether the languages succeed in easing the development of safe code, with a particular focus on each language's compile-time features and memory management techniques. C++, D, and Rust are evaluated on both subjective and empirical criteria. In order to evaluate the success of each language's design goals, I have implemented a number of small programs that demonstrate common tasks in systems programming, each in C++, D, and Rust. I recruited a number of volunteers with prior experience with C++ to attempt the implementation of these programs in D or Rust as well. Each volunteer documented his or her development process in detail, particularly noting any errors or bugs that were encountered. The programmers tallied and categorized each error. This data was used to analyze whether a particular language makes it easier to avoid certain errors. I then evaluated each language on expressiveness and ease of development to determine whether the language's design goals have been met.

## Acknowledgments

The author is deeply thankful to Professor Kim Bruce for his help through every step of writing this document. He is grateful to Sam Posner and Andrew Fishberg as well for their inputs on determining the set of programs for the study. Additional thanks go to the volunteers who dedicated their time and effort to the study. And lastly, a special thanks to the author's parents for their unwavering support and advice in whatever his endeavors.

# Contents

Abstract . . . . . i
Acknowledgments . . . . . iii
List of Figures . . . . . vii
List of Tables . . . . . ix
List of Listings . . . . . xi

**1 Introduction** **1**

**2 Background** **3**
2.1 History . . . . . 3
2.2 Macros and Conditional Compilation . . . . . 5
2.3 Memory Management . . . . . 9

**3 Evaluation Strategy** **15**
3.1 Criteria . . . . . 15
3.2 Experimental Design . . . . . 17

**4 Results** **21**
4.1 Experimental Results . . . . . 21
4.2 Discussion . . . . . 23
4.3 Evaluation . . . . . 25

**5 Conclusion** **27**

**Bibliography** **29**

**A Volunteer Resources** **31**
A.1 Project Volunteer Information . . . . . 32
A.2 D Resources . . . . . 39
A.3 Rust Resources . . . . . 40

**B Solutions** **43**
B.1 Hello, World . . . . . 44
B.2 Sentence Splitter . . . . . 44
B.3 Integer Linked List . . . . . 49
B.4 Generic Array List . . . . . 57
B.5 Parallel Merge Sort . . . . . 63
B.6 Brainfuck Interpreter . . . . . 67

# List of Figures

4.1 Sample error log . . . . . 21

# List of Tables

4.1 Sentence splitter error log summary . . . . . 22
4.2 Integer linked list error log summary . . . . . 23
4.3 Generic array list error log summary . . . . . 23
4.4 Parallel mergesort error log summary . . . . . 23
4.5 Brainfuck interpreter error log summary . . . . . 24

# List of Listings


1 The C macro preprocessor . . . . . 5
2 Macro side-effect evaluation (bug) . . . . . 6
3 C++ header file and definition . . . . . 7
4 Conditional compilation in C++ . . . . . 7
5 Conditional compilation in D . . . . . 8
6 Conditional compilation in Rust . . . . . 9
7 C memory management . . . . . 10
8 Primitive C++ memory management . . . . . 10
9 Modern C++ memory management . . . . . 11
10 D scope guards . . . . . 12
11 C++ use of moved value (bug) . . . . . 13
12 Rust use of moved value (compilation error) . . . . . 13

# Chapter 1

# Introduction

Systems programming is an extremely important part of computing today. The raw speed and low-level access to hardware provided by systems programming languages are necessary for embedded systems, networking, and gaming, plus any number of other applications. However, such command of a computer's hardware naturally invites danger[1].

Mistakes in managing memory can lead to run-time crashes or security vulnerabilities such as buffer overflow or format string attacks [SZ12, Sea13]. While there are many tools designed to allow programmers to catch such errors, the ideal solution would be to eliminate the burden of manual memory management altogether. Other languages such as Java and C#, inspired by this goal, have removed that need. However, due to their dependence on a virtual machine, they have sacrificed performance and the ability to interface directly with hardware [Ale10]. The languages examined in this paper, D and Rust, do not use a virtual machine, instead opting for native compilation. The languages aim to match the low-level speed and power of C++ while attempting to make code easier to write both from an expressiveness and correctness standpoint.

This paper analyzes the Rust and D programming languages from as objective a standpoint as possible. I primarily focus on the features that each language makes at compile time, with memory-safety and preprocessing features taking a particular emphasis, due to the misuse of these features causing dangerous errors in C++.

[1]Throughout this paper, when I refer to a concept or feature as "dangerous," I mean that it is error-prone or difficult to reason about, especially if the errors that may arise from its use are resistant to debugging.

# Chapter 2

# Background

> "D was conceived by people who were tired of how clunky C++ is. Rust was conceived by peoples who were tired of how unsafe C++ is."
>
> *User kibwen*
> *Hacker News*

In order to adequately discuss the design decisions that the implementors of Rust and D have made, it is important to determine a context from which we can compare them. Since both languages aim to occupy the same space as C++, I have decided to compare the languages using C++ as a reference point. In addition to considering the historical context that each language developed in, I focus on a few important features that are essential to systems programming.

## 2.1 History

Though C++, D, and Rust are all related, the languages developed within different historical contexts.

C++ was developed in the early '80s by Bjarne Stroustrup. Stroustrup wished to bring high-level features such as classes, strong typing, and default arguments to C. It attempts to provide the programmer with a one-to-one mapping of built-in types to the hardware, while also offering flexible abstractions to allow user-defined types to take advantage of the same facilities available to the built-ins. Stroustrup himself outlined the design philosophy of C++ in two points:

- *Leave no room for a lower-level language below C++.*
- *What you don't use you don't pay for.* This is also known as the "zero-overhead principle".

These principles serve to keep C++ close to its roots in C while maintaining the abstractions that make it a high-level language [Str13]. However, C++ is often criticized for its complicated and confusing behavior in situations such function overload resolution [Dew05]. Despite its problems, C++ remains one of the most popular languages in the world.

D was created in 2001 by Walter Bright, and later developed by Digital Mars, a compiler development company [Mar14a]. As the name indicates, D has drawn much inspiration from C++, as C++ did from C. In Bright's own words, he wished to design a language as "it should be done" [Ale10]. To this end, D has made a number of backwards-incompatible changes from C++. The D website cites a number of reasons why D is necessary. Most relevant to this paper are its assertions about the inherent complexity in C++ due to the sheer number of features present in the language, the burden of explicit memory management, the difficulty in tracking down pointer bugs, and the hindrance of backwards compatibility with C [Mar14a]. D is often compared to C++: it is commonly thought of as a "better C++" [Ale10]. While less popular than C++, D maintains a healthy online presence and is promoted by C++ gurus such as Andrei Alexandrescu and Scott Meyers [Ale10].

Rust is a quite new programming language developed primarily by Mozilla starting in 2012. Rust aims to give programmers enough power to access the computer's hardware while providing "strong guarantees about isolation, concurrency, and memory safety" [MK14]. Syntactically, Rust is a more radical departure from C++ than D. However, it has similar design goals. It aims to be used in such performance-critical applications as operating systems and web browsers[1]. Despite not having a stable release at the time of this writing, Rust generates a considerable amount of discussion and written code, as evidenced by the over one million downloads served by `crates.io`, Rust's package hosting website [Rus14a].

[1] Rust development is driven in particular by the Servo Project (`https://github.com/servo/servo`), an experimental web browser. Servo requirements have inspired a number of Rust features.

## 2.2 Macros and Conditional Compilation

Code generation is an extremely important feature of systems programming languages. Because low-level code is very dependent on the underlying hardware, different code must be generated for different platforms. Thus, C++, D, and Rust all have their own methods for generating code at compile-time.

When compiling C++, code is first passed through the macro preprocessor, which performs textual replacements and substitutions depending on the instructions provided to the processor. These instructions are signaled by the presence of a '#' at the beginning of the line. Listing 1 demonstrates a simple preprocessor directive. Without the preprocessor, the code snippet would fail to compile due to an undefined identifier `THREE`. The preprocessor replaces all occurrences of `THREE` with the integer literal '3', allowing the assertion to succeed [Ols13].

```
#define THREE 3

assert(THREE == 3);
```

Listing 1: The C macro preprocessor

Another use of the preprocessor is to generate code that would be tedious to write out by hand. The `max` function can be implemented as a macro that expands `max(a, b)` to `a > b ? a : b`. A macro is advantageous to a function because a function call introduces much more overhead than a conditional. While this implementation works in many cases, consider the code in Listing 2. On first glance, one might expect the code to return 2, as `x` is preincremented, making it greater than `y`. The value would then be returned. Since macro expansion is simple textual replacement, the expression is actually evaluated twice, meaning that the value returned from `max` is actually 3 [Fou15]. Such bugs are very subtle and difficult to avoid.

```
#define max(A, B)  ((A) > (B) ? (A) : (B))

int x = 1;
int y = 1;

int max = max(++x, y);    // Uh oh...

// Expands to... (parentheses removed for clarity)
int max = ++x > y ? ++x : y;
```

Listing 2: Macro side-effect evaluation (bug)

Despite this danger, the preprocessor is necessary to expose an interface of a class or library by using the `#include` directive to insert the contents of one file (the "header") within another. The implementation is kept inside a single source file, and any number of other source files may use the interface's implementation. Listing 3 contains a sample header file[2]. This strategy works because C++ allows a programmer to separate declaration from definition. The header file contains declarations, which define the interface of a class or function. Another file contains the definition, or implementation, of the interface. Because a symbol must be defined and declared once, programmers must use "include guards" to ensure this constraint holds. Include guards work by checking if a certain preprocessor symbol is defined, which is usually based on the name of the file being included. If the symbol is defined already, no text is included. Otherwise, the symbol is defined to prevent subsequent inclusion and the text of the header is included. A header file may be included multiple times as long as there is only one definition. Since C++ does not support modules, this use of the preprocessor is required to support multi-file programs.

[2]The data structures in Appendix B also demonstrate the use of headers.

```
#ifndef HEADER_HPP
#define HEADER_HPP

int foo(int a, int b);

#endif
```

```
#include "header.hpp"

int foo(int a, int b) {
    // ...
}
```

Listing 3: C++ header file and definition

The preprocessor is also necessary for conditional compilation, where different code should be compiled depending on various conditions such as the underlying architecture or character-encoding support. For example, the code in Listing 4 will store different values in the `OS` variable depending on whether the software was compiled on Windows or another operating system.

```
#if defined(WIN32) || defined(_WIN32) || defined(__WIN32)
static const std::string OS = "Windows";
#else
static const std::string OS = "Unix";
#endif
```

Listing 4: Conditional compilation in C++

Achieving conditional compilation in this way is unfortunately an error-prone practice. Consider the situation where there is a typo (such as a missing double-quote) in the Windows branch, and compilation is attempted on Unix. The compilation would succeed, because the compiler would never see the Windows-specific branch of code after preprocessing. However, if compilation were attempted on Windows, compilation would fail. Unfortunately, the C macro preprocessor is the only way to obtain this result. Stroustrup advises minimizing the use of the C++ preprocessor, and that it should only be used for the specific cases of conditional compilation and header files [Str13].

D provides safer alternatives to the C++ preprocessor. For example,

inclusion is handled by modules, which safely manage imports of symbols from other files, as seen in the data structures in Appendix B. This also removes the need for include guards, because each symbol is guaranteed to be imported only once from a module by the compiler. The interface is inferred from the function definitions in the module, so there is no need to copy the interface, like the redundant function signature in Listing 3.

In D, conditional compilation is achieved through the `version` keyword [Arm07]. Listing 5 demonstrates how D would implement the same code as Listing 4. Since the `version` keyword is part of the language grammar, syntax errors will be caught by the compiler regardless of what compile path the program requires. D also contains an aggressive inliner, which removes the need to implement small functions as macros [Mar14c]. Instead, the compiler will directly insert the function body into the caller's code, meaning that the compiled code has no function call at all.

```
version (Windows) {
    static const string OS = "Windows";
} else {
    static const string OS = "Unix";
}
```

Listing 5: Conditional compilation in D

Like D, Rust also has a module system to handle inclusion and multi-file compilation, so there is no need for a preprocessor. However, unlike D, Rust has a macro system. Rust macros are far more powerful than C++ macros. Rust macros operate on the abstract syntax tree of the program, rather than on the text. This allows macros to be type-safe and extend the language itself. In Rust, `println` is a macro that performs type-checking on its arguments to ensure that the format string contains the proper number of format placeholders for the arguments. In other languages, it might be possible to check this with a special code path in the compiler, but in Rust the language itself does the checking.

Rust also has a safer alternative for conditional compilation. In Rust, one can write conditional code by applying attributes to arbitrary functions or symbols, as seen in Listing 6. Like D, since configuration attributes are part of the language, there is no worry that syntax errors will arise when compiling on different platforms.

```rust
#[cfg(target_family = "windows")]
const OS: &'static str = "Windows";

#[cfg(target_family = "unix")]
const OS: &'static str = "Unix";
```

Listing 6: Conditional compilation in Rust

## 2.3 Memory Management

Memory management is a key component of systems languages. Many performance-critical applications depend on the power of being able to manage memory manually. However, this opens up an entire class of bugs. Some of these bugs include memory leaks (forgetting to free allocated memory) and use-after-free (using a pointer to memory that has already been reclaimed by the memory manager). These bugs are often difficult to track down.

In C++, all objects have a constructor and a destructor, which are functions that allocate and deallocate any memory needed by the object respectively. If the object is allocated on the stack, the constructor is run when the object is created, and the destructor is run when the object goes out of scope (the end of the innermost block). Stack memory is relatively easy to reason about, but the programmer must be careful to avoid leaking a reference or pointer to a stack-allocated variable outside of the scope where the variable was defined. This could lead to undefined behavior.

Undefined behavior is characterized by the lack of a definition of what an implementation of C++ should do in a given situation [iso11]. In other words, if a program incurs undefined behavior, then an implementation is in no way required to act in any particular way. The program may not compile, may crash at run-time, or anything else. Clearly it is impossible to reason about the correctness of a program that invokes undefined behavior, so C++ programmers must avoid it at all costs.

C++'s heap memory management works similarly to its ancestor, C. In fact, due to to its commitment to backwards compatibility, C++ even inherits the `malloc()` and `free()` functions, as seen in Listing 7. These functions allow the programmer to request and return blocks of memory to the free store, or heap. However, C++ programmers should never use these functions, instead opting for the `new` and `delete` operators, which ensure that the correct amount of memory is allocated and deallocated and that

constructors and destructors are run. C++ has additional operators for array allocation: `new[]` and `delete[]`. Listing 8 uses the operators to allocate an array. These keywords are slightly safer than `malloc()` and `free()` because the programmer no longer has to keep track of the exact size of the allocation. However, the programmer must still remember to to `delete` all memory acquired by `new` to avoid memory leaks, and refrain from using deallocated or unallocated memory to prevent undefined behavior [Str13]. In addition, the correct form of the keyword must be used, as using the non-array `delete` may leave the heap in an inconsistent state.

```
/* Allocates 10 elements on the heap. */
int* elements = malloc(sizeof(int) * 10);

/* ... */

/* Release the memory for reuse. */
free(elements);
```

Listing 7: C memory management

```
// Allocates 10 elements on the heap.
int[] elements = new int[10];

// ...

// Release the memory for reuse.
delete[] elements;      // Don't forget the '[]'!
```

Listing 8: Primitive C++ memory management

C++ programmers are well-aware of the problems that arise from using `new` and `delete` improperly. In fact, Stroustrup warns that "naked `new`" (using `new` to allocate an object directly) ought to be avoided. Instead, he advises programmers to use stack-based allocation when possible, and in other cases use manager objects such as `unique_ptr` and `shared_ptr`[3]. This idiom is known as "Resource Acquisition Is Initialization", or RAII [Str13]. These containers help abstract memory management away from the programmer by ensuring `delete` is called when the pointer containers go out of scope by invoking the keyword in their destructors.

[3]These containers were introduced in C++11.

```cpp
#include <array>
#include <memory>

// Allocates 10 elements on the heap.
std::unique_ptr<int[]> elementsPtr(new int[10]);
// Using std::array avoids explicitly using 'new'.
std::array<int, 10> elementsArray;

// ...

// Memory is freed when the unique_ptr and array go out of
// scope.
```

Listing 9: Modern C++ memory management

D's creators acknowledged the problems with manual memory management and opted to remove the need for it entirely. D handles memory management through a garbage collector. D classes are automatically allocated on the heap, and all other data is created on the stack. The garbage collector frees any memory that has gone out of scope (though not necessarily immediately after). This removes the need for the programmer to explicitly allocate and deallocate memory through `new` and `delete`. Like C++, destructors are executed when variables go out of scope, allowing RAII behavior. In addition, D provides the "Scope Guard" statement, which offers more fine-grained control over when scope-dependent blocks of code should execute, seen in Listing 10. For example, if an exception is not thrown, the code would print `success` and then `exited` on their own lines. If an exception was thrown, the code would print `failure`, then `exited`. A possible use of this feature would be to guarantee that resources are closed properly when runtime exceptions are encountered.

```d
import std.stdio;

try {
    scope(exit) writeln("exited");
    scope(failure) writeln("failure");
    scope(success) writeln("success");
    // Code that may throw an exception...
} catch (Exception e) {}
```

Listing 10: D scope guards

D programmers may also opt-out of the garbage collector by marking functions with the `@nogc` attribute. Within `@nogc` functions, the garbage collector will not run. This feature is intended for situations where the overhead introduced by the D runtime (which contains the garbage collector) is unacceptable, such as when implementing a kernel or in performance-critical applications. However, this introduces a number of problems, as allocating classes and other types that rely on the garbage will be no longer possible without using the C++-like `new` and `delete` operators. These operators bring the same problems that they had in C++, and D has no equivalent to `unique_ptr`, making their use arguably more dangerous in D. In addition, some parts of the standard library perform heap allocations, making it impossible to use these functions in some applications. The goal is to eventually mark all of D's standard library as `@nogc`.

Rust, on the other hand, attempts to find a middle ground between the memory-unsafe performance of C++ and the safer, but slower overhead of D. One of the interesting features of Rust is its guarantees about heap access. One could think of Rust as implicitly creating every variable as a C++-style `unique_ptr`. It accomplishes this by strictly enforcing the concept of ownership through the type system and a "borrow checker". Ownership means keeping track of which objects or functions "own" pieces of memory. For example, passing a unique pointer to a function means that that function is "borrowing" the memory pointed to by the pointer. If we assume that the caller owns the memory, then the function should return the memory to the caller in a consistent state[4]. Since C++11, it is possible to explicitly declare the programmer intent for the owners of pointers. C++ pointer containers include `unique_ptr`, `shared_ptr`, or `weak_ptr`. For example, a `unique_ptr` is the only owner of its memory. It enforces this by disallowing copying and requiring that reassignment be done through a move, which transfers

[4]That is, not deallocating or moving it.

ownership of a piece of memory from one object or scope to another. Moves are far more efficient than copies, but they leave the previous owner in an inconsistent state. This ownership transfer may introduce bugs, as shown in Listing 11.

```cpp
#include <memory>
#include <iostream>

// Allocate an int on the heap
std::unique_ptr<int> movedPtr(new int(10));

// Change the int's owner to a new pointer.
std::unique_ptr<int> ptr = std::move(movedPtr);

// Attempt to dereference the pointers.
std::cout << *ptr << std::endl;
std::cout << *movedPtr << std::endl;    // Segfault!
```

Listing 11: C++ use of moved value (bug)

Rust avoids this class of bug entirely by guaranteeing at compile time that moved memory cannot be used. Attempting to compile Listing 12 would result in `error: use of moved value: ‘x‘`. Even more powerful than this is that Rust guarantees that "no other writable pointers alias to this heap memory", meaning that it is impossible for multiple objects to write to the same memory location (unless the programmer were to use an `Rc` pointer, which allows multiple readers and writers through reference counting) [Rus14c].

```rust
let x = Box::new(5i);        // Allocate an int on the heap.
let y = x;                   // Change the int's owner to y.
println!("{}", x);           // error: use of moved value: 'x'
```

Listing 12: Rust use of moved value (compilation error)

# Chapter 3

# Evaluation Strategy

## 3.1 Criteria

One of the most challenging aspects of evaluating languages is deciding the criteria on which they are evaluated. In order to obtain as comprehensive an evaluation of Rust and D as possible, I wish to include both qualitative and quantitative criteria.

AlGhamdi and Urban provide an excellent list of qualitative methodologies on which I have based my own methodology [AU93]. Their paper summarizes twelve methodologies, and the factors that may be used to achieve an apt comparison. The following list summarizes the methodologies expounded in their paper that I will employ in my project.

Other papers have attempted a feature-based comparison [LRS+03], with no evaluation. Still others have performed such a comparison and then evaluated the features within different domains [FG82]. I performed a feature-based comparison in Chapter 2. I chose not to evaluate the languages based on these features alone due to focusing on the general domain of systems programming. Instead, my evaluation remains at a language-level rather than feature-level.

1. Comparison of Philosophy and History

   Since both Rust and D occupy a similar domain, the main factor that I will use to distinguish the two is the "intention of [the] designers". While these languages were created nearly a decade apart, differ in their corporate affiliations, and have varying development team sizes, I find these factors less relevant for my study. I believe that the only factor that deeply affects programmers using D and Rust is the design philosophy behind the languages.

2. The Degree of Permissiveness of the Language

   This methodology includes criteria such as the ability of the programmer to circumvent type-checking, operator overloading, and run-time checking. I am particularly interested in the memory safety of Rust and D compared to C++. While these languages will allow a programmer to circumvent the type system or perform unsafe memory accesses, such practices are discouraged. So, I am interested in the effectiveness of each language in avoiding the requirement of such unsafe techniques.

3. Language Contributions to Program Readability

   It is often said that code is read far more than it is written. In large systems that depend on reliability, this is surely true. It follows that if code is easier to read, it is easier to locate defects or bugs.

4. Language Contributions to Program Reliability

   This is perhaps the most important methodology to my project. I am particularly interested in examining how Rust and D attempt to avoid the common pitfalls that plague C++ development. These include but are not limited to uninitialized variables, null pointers, and memory leaks.

5. Data Structuring Facilities

   This methodology is also quite important to my study. Rust and D are both strongly typed, and each language provides a number of ways to inform the compiler of programmer intent for the usage of various data types. Exploring the numerous primitive data types, and the ability to construct new types from those primitives is integral to understanding the power of each language.

6. Control Facilities

   I am interested in procedural-level control facilities. This includes parameter-passing methods, concurrency, and generics/templates.

7. Language Contributions to Program Cost

   This methodology explores the various costs involved with writing programs in a language under consideration. This involves the cost of learning the language, the cost of writing a program in the language, and even the cost of compiling a program. The cost of executing and maintaining a program less relevant to this paper, but I imagine

the maintainence cost is somewhat encompassed by the criteria for evaluating the languages' readability. While this methodology is particularly important for new programmers, and becomes less apparent with experience, it is nonetheless important to consider.

In contrast to the methodologies listed above, I found a number of methodologies irrelevant. For example, D and Rust appear almost identical in terms of modularity ("Language Contributions to Program Modularity"), portability ("Portability of a Language"), I/O facilities ("Input/Output"), and support for inline assembly and foreign functions ("Escape from a Language"). These methodologies either contribute little to my goal of studying the memory safety of Rust and D, or the features provided by each language are so similar that there is little comparison to be made between the two.

## 3.2 Experimental Design

While I believe that the aforementioned methodologies comprise a satisfactory qualitative evaluation strategy for Rust and D, I do not believe that a full comparison can be achieved with these methodologies alone. Furthermore, many of the criteria are rather subjective. For my project, I have strived to develop a method which can be used to compare various language features while avoiding subjectivity or bias.

Originally, I planned to come up with a number of programs that embodied the core of systems programming, and then develop these programs in C++, D, and Rust. Then, using my own experiences, I would attempt to evaluate the effectiveness of each language for developing these programs. However, this approach is flawed in a number of ways. For example, suppose I had attempted to implement a program with logic that required a large amount of conditionals in Rust. I would likely have encountered a number of bugs with my initial implementation but eventually have come up with a satisfactory result. Then, upon moving onto D, I would have avoided most of the errors that I encountered while working on the Rust implementation. My development process in D would likely have felt more natural, and the code would probably look eerily like Rust. To mitigate this problem, I instead recruited a number of volunteers to learn each language individually, and implement a series of programs meant to demonstrate various features that are important to systems languages.

I advertised my study both on the Computer Science Facebook group and through Computer Science colloquium. Once volunteers indicated their

interest, I sent them a document[1] detailing my expectations for the project. The volunteers then sent me information including their name, major, experience with programming, and a confirmation that they read the document in its entirety.

The document contained resources for installing and using the languages. In order to reduce bias introduced by outside resources, I limited the resources available to each volunteer to the language reference and an additional book. Using websites such as StackOverflow was explicitly disallowed. I also disallowed installing autocomplete packages or IDEs.

Each volunteer was then assigned either Rust or D as a language. I tried to satisfy the volunteer's preference for which language to learn, though I first ensured that the experience level between each language would be comparable. Over the next month, each volunteer was expected to attempt to implement one of five programs. The volunteers were not expected to spend more than 2 hours on each program, but most volunteers who finished worked on each program to completion.

The programs implemented by the volunteers were designed to cover an assortment of language features as well as to highlight common bugs. The programs were as follows:

1. Sentence Splitter

   Volunteers were asked to implement a single string into sentences using a set of heuristics. The difficulty of this program stems from the fact that while sentences are delimited by periods, question marks, and exclamation points, they may also include websites, titles, and abbreviations, all of which are *not* boundaries.

   This program was meant to introduce the programmers to string manipulation techniques and required a large amount of logical branching.

2. Integer Linked List

   The next program was to implement a simple linked list data structure that operates on integers. A number of operations were required to be supported, including insertion, deletion, retrieval, and methods to query the size, head and tail of the list.

   This program introduced the volunteers to object-orientation, pointer manipulation, and memory allocation.

3. Generic Array List

[1] The document may be found in Appendix A.

The next program involved creating an array list that supported generic elements. The array list was required to support the same operations as the integer linked list, and covered the same features of the languages.

4. Parallel Mergesort

   Volunteers were then asked to implement the parallel mergesort algorithm. This algorithm is relatively simple to understand, and easily parallelizable.

   This program introduced the programmers to recursion and concurrency.

5. Brainfuck Interpreter

   Lastly, the programmers were asked to implement a Brainfuck interpreter. Brainfuck is an esoteric programming language, known for being fiendishly unreadable. However, the language's semantics are quite easy to understand. In short, there are eight meaningful characters in the language, all of which either manipulate the "data pointer" in some way (by incrementing, decrementing, etc.), or perform I/O on the byte pointed at by the pointer.

   This program served as a kind of "capstone" for the study. It is larger than the other programs, and involves file I/O and pointer manipulation.

Each volunteer was also required to document their development process. While developing each program, the volunteer would note every error encountered, categorize it as a syntax error, logic error, or resource error, and whether the error was caught at compile-time or run-time.

# Chapter 4

# Results

Given Rust's focus on compile-time error catching and D's focus on ease of use, I hypothesized that my volunteers would encounter a much greater number of compile time issues with Rust, and that D programmers would find that their programs ran into more runtime issues. Also, D's syntax would be more familiar to my volunteers. I found that my hypothesis was true.

## 4.1 Experimental Results

I was able to recruit seven volunteers that generously offered their time and effort to my project. Each volunteer was very experienced in programming: six were computer science majors, and one was a computer science minor. Figure 4.1 contains a snippet of an error log submitted by one of my volunteers.

```
Syntax error - Wrong syntax of println - Compile time
Syntax error - Wrote "splitSentences" instead of "split_sentences" -
    Compile time
Syntax error - Used old variable name - Compile time
Syntax error - Tried to iterate over len() instead of range(0, len())
    - Compile time
Syntax error - Used one argument instead of two with range() - Compile
     time
Syntax error - Passed char instead of &char in the contains func -
    Compile time
Syntax error - Used [char] instead of &[char] in initialization of
    array - Compile time
```

Figure 4.1: Sample error log

Unfortunately, due to time constraints, many volunteers were unable to follow through with the entire study. I have obtained results from those who were able to attempt each program, though by the fourth program I only had volunteers who were writing D left.

All of my volunteers were able to complete the sentence splitter. It was very interesting to see the types of errors that were encountered during this phase of the experiment, as it was the volunteers' first introduction to the language. As expected, Rust programmers found many more compile time issues. Somewhat surprisingly, D programmers ran into segmentation faults due to the behavior of attempting to access iterators out of bounds, while Rust programmers simply encountered panics[1]. Though this program required no advanced language features, there was a fair amount of difficultly in ensuring that arrays were not accessed invalidly. The errors are summarized in Table 4.1.

| Volunteer | Syntax | Logic | Resource | Notes |
|---|---|---|---|---|
| A (D) | 6 | 2 | 0 | Does not compile |
| B (Rust) | 6 | 0 | 0 | Did not finish |
| C (D) | 8 | 2 | 0 | Passes test suite |
| D (D) | 2 | 1 | 0 | Infinite loop on ellipsis |
| E (Rust) | 18 | 5 | 0 | Passes test suite |
| F (Rust) | 15 | 0 | 0 | Passes test suite |
| G (D) | 4 | 2 | 0 | Passes test suite |

Table 4.1: Sentence splitter error log summary

Four volunteers attempted the integer linked list. Volunteer B was particularly frustrated by this assignment, as it was extremely difficult to implement a true linked list in Rust without resorting to language facilities guarded by the `unsafe` keyword. While it is not discouraged to use such features, for a beginner they are understandably intimidating. The results from this program are located in Table 4.2.

The generic array list was easier due to only requiring the ownership of a single pointer. However, this program was also difficult to implement in Rust due to requiring dynamic allocation of memory. There is currently no safe way to allocate an array of a generic type, so the only way to complete the program without resorting to unsafe allocation is to use `Vec`, which is a dynamic array in itself. This might be considered against the spirit of the

[1] Similar to an uncaught exception.

| Volunteer | Syntax | Logic | Resource | Notes |
|---|---|---|---|---|
| B (Rust) | 10 | 0 | 0 | Did not finish |
| C (D) | 9 | 0 | 1 | Passes test suite |
| D (D) | 1 | 0 | 3 | Segfault on `remove()` |
| G (D) | 6 | 1 | 2 | Passes test suite |

Table 4.2: Integer linked list error log summary

exercise.

| Volunteer | Syntax | Logic | Resource | Notes |
|---|---|---|---|---|
| B (Rust) | 1 | 0 | 0 | Did not finish |
| D (D) | 2 | 0 | 0 | Passes test suite |
| G (D) | 4 | 0 | 0 | Passes, `head()` and `tail()` not implemented |

Table 4.3: Generic array list error log summary

Unfortunately, at this point in the study I no longer had any volunteers on the Rust side. However, the D volunteers had little trouble implementing the mergesort algorithm. Surprisingly, this ended up being the easiest task to implement.

| Volunteer | Syntax | Logic | Resource | Notes |
|---|---|---|---|---|
| D (D) | 3 | 1 | 0 | Passes test suite |
| G (D) | 3 | 2 | 0 | Passes test suite |

Table 4.4: Parallel mergesort error log summary

No volunteer was able to correctly implement the brainfuck interpreter. Both programs either output garbage or triggered a segmentation fault.

## 4.2 Discussion

Based on my own experiences with the implementation of my programs, and the experiences of the volunteers of my study, I have assessed both Rust and D as productive languages. I explore the strengths and weaknesses of both languages through the context of each of the programs.

| Volunteer | Syntax | Logic | Resource | Notes |
|---|---|---|---|---|
| D (D) | 9 | 0 | 0 | Segmentation fault |
| G (D) | 4 | 1 | 0 | Fails test suite |

Table 4.5: Brainfuck interpreter error log summary

Since the error logs are self reported, it would not be productive to form an analysis based on the numbers themselves. However, it is possible to observe some overall trends, and draw conclusions based upon those trends and my own experiences with the implementation of the programs of my study.

For the sentence splitter, the algorithm that I chose to implement for my solution depends on manipulating a pointer to the string of data that contains the paragraph to split. Due to the pointer manipulation, it was easier to use D for this algorithm. However, it was very easy to encounter range violations in D, which is a runtime error. Since my Rust implementation used Rust's iterators, it was easy to verify that I wasn't iterating over the same characters multiple times.

The data structures offered some very interesting data. Rust in particular offered some real challenges. I sympathize with the Rust volunteer who struggled with implementing the linked list. Though Rust does offer the programmer the ability to venture into memory unsafe territory, it is hard for a beginner to determine when this is the correct choice. While Rust is permissive enough to allow this, it was surprising that such a common operation as allocating memory would need to resort to unsafe code.

The Rust volunteer that completed these structures was very frustrated by the generic array list as well. This is due to the fact that it is very difficult to implement these structure without resorting to `unsafe` facilities of the language. Thankfully, Rust provides dynamic arrays and linked lists in the standard library, but I imagine that it would be very difficult to implement data structures that require pointer manipulation or cycles as a beginner, which is to the language's detriment.

The parallel mergesort was an especially successful exercise. Both volunteers were able to complete it, and the errors that they encountered were indicative of the problems that are difficult to avoid in D. Most of the errors encountered were syntax errors, but the logic errors that arose were range violations, which were particular to D. However, I expect that if more Rust volunteers had completed this program I would have encountered a similar distribution of off-by-one errors in vector accesses.

I was surprised with the results of the Brainfuck interpreter. Each volunteer completed the program, but the programs were not correct. The "Hello World" program used to test the interpreters was meant to expose bugs in interpreters, so perhaps the programs worked as expected on simpler inputs. This example did not expose any particular differences between the languages, as I had a similar experience in implementing the interpreter in both languages.

Overall, it was interesting to see how beginners approached the languages differently. While some volunteers embraced the idioms of the new languages, some volunteers attempted to code Java- or C++-like code, which could have hindered them.

As for the study itself, there were a few things that I would have changed. I was disappointed with the attrition of volunteers that the study suffered as the semester got busier. Other studies have permitted volunteers to leave tasks unfinished [HH13], but these studies had a much larger sample size and longer time frame. In a survey that I distributed after the study, some volunteers expressed that they would have appreciated more direction in their objectives. I originally wanted to give the volunteers as much freedom as possible in their schedule, but this made it difficult for some to stay on track.

One volunteer suggested that I hold "lab sessions", or scheduled a specific time and place for the volunteers to work on each program. I think that this would have been beneficial in helping the volunteers stay on schedule, as well as allowing me to more closely monitor the error logging process. However, this might have introduced additional overhead to resolve the volunteers' schedules. Another volunteer proposed that more starter code be given out, as this would cut down on the implementation time for each task.

The volunteers generally liked the languages they were assigned and would be inclined to use them for personal projects. The reception towards D was warmer due to its similarity to C++ and Java, languages with which my volunteers were more familiar.

## 4.3 Evaluation

Both D and Rust are improvements over C++. The increased power of their respective type systems, lack of required manual memory management, and compile-time features are attractive enough to warrant creating new programs in these languages. However, I believe that they occupy different needs.

D definitely achieves its historical goal of being an easier-to-use C++. Its syntax obviates the need for many tedious-to-write constructs in C++. Less syntax generally meant more readable code. However, I found the unhelpful error messages a hindrance to development. I often had to open `gdb` to debug segmentation faults and range violations, which, in addition to being runtime errors, did not give line numbers in their error messages. If I needed to develop a fast prototype, I would consider using D.

This goes along with the fact that D is generally more permissive than Rust. D will let you write more constructs that could lead to runtime issues. This permissiveness is also is seen in the languages' policies towards mutability. While D programmers must opt into immutability by using the `const` or `immutable` keyword, Rust programmers must opt into *mutability* by using the `mut` keyword. There are a number of optimization opportunities that are available to the compiler with immutable data, and the fact that Rust encourages immutability improves the chances that these optimizations can be made. It also forces the programmer to think about whether variables actually need to be mutable. Though Rust is less permissive in this regard, I think it is a huge boon to the language.

As a beginner I found it very difficult to rapidly develop programs in Rust. However, the error messages were far more helpful than D, and the compiler disallowed constructs that I originally attempted because they were error-prone. I also found its syntax more readable. Not only are parentheses optional around expressions in block statements, the compiler actually emits a warning if they are present. This, along with other lint warnings emitted by the compiler ensure that Rust code has a consistent, readable style which contributes to the readability of arbitrary Rust code seen for the first time.

I believe that both D and Rust have the potential to displace C++ as a systems language of choice. Rust is perhaps more useful for safety-critical applications, while D would be more useful for rapid development.

# Chapter 5

# Conclusion

In this paper I have discussed and evaluated the D and Rust programming languages with a focus on productivity for beginners. I believe that both languages accomplish their design goals: D to make a more ergonomic C++, and Rust to bring memory safety to the forefront of a programmer's thought process.

I examined the history and design philosophies of both languages and compared features that highlighted the deficiencies of C++ as a productive, safe language. I found that both D and Rust had sensible and useful alternatives to the C++ feature under scrutiny.

I also performed a user study to examine how experienced C++ programmers used the new languages. My study, though suffering from a lack of data, confirmed my hypothesis that D would be easier to learn and develop into, though Rust would assist the programmers in avoiding bugs.

There are many factors that determine what the true "successor" of C++ will be. Considering the amount of legacy code in production today, I doubt that C++ will ever be truly succeeded. However, I am confident that both D and Rust have their place in software development in the future.

# Bibliography


[Ale10] Andrei Alexandrescu. *The D Programming Language*. Addison-Wesley Professional, 1st edition, 2010.

[All10] Charles D. Allison. D: A programming language for our time. *J. Comput. Sci. Coll.*, 26(2):113–119, December 2010.

[Arm07] Ameer Armaly. The D Programming Language. *Linux J.*, 2007(155):9–, March 2007.

[AU93] Jarallah AlGhamdi and Joseph Urban. Comparing and assessing programming languages: Basis for a qualitative methodology. In *Proceedings of the 1993 ACM/SIGAPP Symposium on Applied Computing: States of the Art and Practice*, SAC '93, pages 222–229, New York, NY, USA, 1993. ACM.

[Dew05] Stephen C. Dewhurst. *C++ Common Knowledge: Essential Intermediate Programming*. Addison-Wesley Professional, 2005.

[FG82] Alan R. Feuer and Narain H. Gehani. Comparison of the Programming Languages C and Pascal. *ACM Comput. Surv.*, 14(1):73–92, March 1982.

[Fou15] The Free Software Foundation. Macro Pitfalls, 2015.

[HH13] Michael Hoppe and Stefan Hanenberg. Do developers benefit from generic types?: An empirical comparison of generic and raw types in java. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '13, pages 457–474, New York, NY, USA, 2013. ACM.

[iso11] Standard for Programming Language C++. Technical Report 14882:2011, 2011.

[LRS+03] H.-W. Loidl, F. Rubio, N. Scaife, K. Hammond, S. Horiguchi, U. Klusik, R. Loogen, G.J. Michaelson, R. Pea, S. Priebe, .J. Rebn, and P.W. Trinder. Comparing parallel functional languages: Programming and performance. *Higher-Order and Symbolic Computation*, 16(3):203–251, 2003.

[Mar14a] Digital Mars. Overview - The D Programming Language. `http://dlang.org/overview.html`, 2014.

[Mar14b] Digital Mars. Programming in D for C++ Programmers. `http://dlang.org/cpptod.html`, 2014.

[Mar14c] Digital Mars. The C Preprocessor vs D. `http://dlang.org/pretod.html`, 2014.

[MK14] Nicholas D. Matsakis and Felix S. Klock, II. The Rust Language. In *Proceedings of the 2014 ACM SIGAda Annual Conference on High Integrity Language Technology*, HILT '14, pages 103–104, New York, NY, USA, 2014. ACM.

[Ols13] Olsson, Mikael. *C++ Quick Syntax Reference*. Apress, 2013.

[Rus14a] Rust Project Developers. The Rust community's crate host. `https://crates.io`, 2014.

[Rus14b] Rust Project Developers. The Rust Guide. `http://doc.rust-lang.org/guide.html`, 2014.

[Rus14c] Rust Project Developers. The Rust Pointer Guide. `http://doc.rust-lang.org/guide-pointers.html`, 2014.

[Rus14d] Rust Project Developers. The Rust Reference. `http://doc.rust-lang.org/reference.html`, 2014.

[Sea13] Robert C. Seacord. *Secure Coding in C and C++*. Addison-Wesley Professional, 2nd edition, 2013.

[Str13] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, Upper Saddle River, NJ, 2013.

[SZ12] Hossain Shahriar and Mohammad Zulkernine. Mitigating program security vulnerabilities: Approaches and challenges. *ACM Comput. Surv.*, 44(3):11:1–11:46, June 2012.

# Appendix A

# Volunteer Resources

The following documents were sent to the volunteers of my project. The documents contain explanations of the project expectations, brief descriptions of the programs to be implemented, resources, and installation instructions for the languages themselves.

The documents were written using Markdown. The versions included in this appendix are rendered from LaTeX generated from the Markdown source. The documents appear online at `https://github.com/euclio/senior-project/tree/master/resources`.

## A.1 Project Volunteer Information

Thank you for volunteering your time to help me with my senior project. In this document I will attempt to cover my expectations from you as well as any resources that you might find useful while working on my project. Please let me know if you have any questions; I'd be happy to discuss the project with you!

### A.1.1 Expectations

Over the course of this project, you will be implementing a series of programs that illustrate various aspects of systems programming: such as memory management, pointer manipulation, and conditional compilation. Each volunteer is expected to have

- Some experience with C++
- Very little experience with either D or Rust

#### Error Logs

For my project, I am interested in the types of errors that are common to each of the languages you will be working in. Specifically, here are the types of errors I am studying:

| Category | Description |
|---|---|
| Syntax error | Any error caused by your program failing to parse. Could be a typo in a variable name, forgetting to put braces, etc. |
| Logic error | Your program parses correctly, but it does not run as intended due to human error. Off-by-one errors are a common example of a logic error. |
| Resource error | The incorrect use of memory management. For example, your program dereferences a NULL pointer or uses deallocated memory. |

In order to obtain data on the types of errors that you run into, I would like you to keep a log of every error you run into while working on each project. Each log should contain the type of error you ran into, a brief description of the error, and whether the error was caught at compile-time or run-time. Here's an example of a log:

| Category | Description | When caught |
|---|---|---|
| Syntax error | Wrote “sring” instead of “string” | Compile-time |
| Logic error | Accidentally iterated one time too many | Run-time |
| Resource error | Used a freed pointer | Run-time |

In short, if the compiler spits out an error, or you have to change your code after you noticed something didn't work, log it! The error logs are key to my study so please take care that they are accurate.

**Rust Note** Since Rust is rather unstable, you might receive warnings about deprecated or unstable features. You may fix these, and you do *not* have to include them in your logs. However, if you are unsure if an error should be included or not, please include it anyways.

### Implementation

Each volunteer will be selected to implement their series of programs in D or Rust.

**Resources** In order to keep the learning environment as free of bias as possible, the resources you may use to learn Rust and D should be limited to the following sites (i.e., no StackOverflow, no IRC, etc.):

**Additional Help** If you would like additional help, you may contact me in which case I will decide to offer help at my own discretion. I will offer help with:

- Algorithmic Questions
- Clarification
- Installing the languages
- Locating information about the languages in the books or docs

I will not:

- Look at your code
- Help you solve compile errors

You are free to use whatever editor you wish. Please install a syntax highlighting package for your editor; I know that packages exist for Sublime Text, emacs, and vim. Please do *not* install any autocomplete packages, as these may skew the results of the study.

**Development** There will be test cases provided in order for you to test your implementation. However, there is no problem if your program does not pass all cases. As long as you spend a significant amount of time on each program (say, 90 minutes), it doesn't matter if your program "works" in the end.

I may use snippets of your code in my final paper, in which case your work will be kept anonymous.

### A.1.2 Installation

It's probably easiest to use your own computer to install each language's compiler. However, if you have a Windows PC, to save yourself some installation headache I recommend either dual-booting or running a VM of a Linux distro such as Ubuntu, or using the lab Macs. I've set up `project` with the needed dependencies of each language as well.

You may find language-specific installation guides in the Rust resources and D resources documents.

### A.1.3 Programming

You will be asked to implement the following programs (listed in order from easiest to hardest).

*Note:* While I don't expect you to handle *all* errors that your program might run into, if you run into exceptional conditions (that is, empty input, invalid indices, etc.), please handle the errors as you feel fit. This could include throwing an exception or returning an `Error` object.

#### Sentence Splitter (source)

Define a function capable of splitting a text into sentences. The standard set of heuristics for sentence splitting includes (but isn't limited to) the following rules:

Sentence boundaries occur at one of "." (periods), "?" or "!", except that

- Periods followed by whitespace followed by a lower case letter are not sentence boundaries.
- Periods followed by a digit with no intervening whitespace are not sentence boundaries.

- Periods followed by whitespace and then an upper case letter, but preceded by any of a short list of titles are not sentence boundaries. Sample titles include Mr., Mrs., Dr., Jr.
- Periods internal to a sequence of letters with no adjacent whitespace are not sentence boundaries (for example, www.aptex.com, or e.g).
- Periods followed by certain kinds of punctuation (notably comma and more periods) are probably not sentence boundaries.

Write a function splitSentences(string input) that, given a string, prints each sentence on a separate line. I am only concerned with the function, so you may pass your test input to the function in any way, whether it be hard-coded, or through stdin, etc.

*Note:* For simplicity, we will assume that we only have one space between each sentence.

### Skills covered

- String manipulation
- Branching

### Integer Linked List

Define a linked list data structure that operates on integers. The linked list should support:

- Insertion in O(n) time.
- Deletion in O(n) time.
- Retrieval in O(n) time.
- A method to retrieve the size.
- A method to retreive the head of the list.
- A method to retrieve the tail of the list.

The IntegerLinkedList class should implement the following Java-like interface:

```
/**
 * A linked list containing only integers.
 */
interface IntegerLinkedList {
  /**
   * Inserts an element into the linked list at the specified index,
   * shifting any elements already in its position down.
   */
```

```
  public void insert(int index, int element);

  /**
   * Removes an element from the specified index, shifting any elements
   * already in the list back up.
   * @return The element that was removed.
   */
  public int remove(int index);

  /**
   * Retrieves an element at the specified position
   */
  public int get(int index);

  /**
   * Returns the number of integers in the linked list.
   */
  public int size();

  /**
   * Returns the first element of the linked list.
   */
  public int head();

  /**
   * Returns the last element of the linked list.
   */
  public int tail();
}
```

### Skills covered

- Object-orientation
- Pointer manipulation
- Memory allocation

### Generic Array List

Implement an array list class (also known as a dynamic array). Unlike the linked list class, this class should support generic elements, like ArrayList in Java. You should use templates (D) or generics (Rust) to implement this class.

The array list should support:

- Insertion in amortized O(1) time.
- Deletion in O(n) time.
- Retrieval in O(1) time.
- A method to retrieve the size.

The ArrayList class should implement the following Java-like interface:

```
/**
 * An array that dynamically resizes as elements are added.
 */
interface ArrayList<E> {
  /**
   * Inserts an element into the array at the specified index,
   * shifting any elements already in the list down.
   */
  public void insert(int index, E element);

  /**
   * Removes an element from the specified index, shifting any
   * elements past its position back up.
   * @return The element that was removed.
   */
  public E remove(int index);

  /**
   * Retrieves an element at the specified index.
   */
  public E get(int index);

  /**
   * Returns the number of elements in the array.
   */
  public int size();
}
```

### Skills covered

- Object-orientation
- Generic programming
- Memory allocation

### Parallel Mergesort

Implement the parallel mergesort algorithm. This algorithm should use std.parallelism (D) or std::thread (Rust). You should also implement a sequential threshold of 10, meaning that if the number of elements in a subarray is less than 10, you should not fork a new thread.

Your function should take in an array.

### Skills covered

- Recursion
- Concurrency

**Brainfsck Interpreter**

Write an interpreter for the brainfsck programming language (thankfully, you are not required to write any brainfsck programs on your own).

Your interpreter should take in a file name as the first argument to the program, and respond to standard input and output as specified in this Wikipedia article. Hint: your interpreter may need to perform multiple passes on the input to handle brackets.

**Skills covered**

- File I/O
- Pointer manipulation

### A.1.4 Completion

Upon completion of each program, you may email your code and error log to me or upload it to a source-hosting website such as GitHub or BitBucket.

### A.1.5 Conclusion

Thank you again for volunteering! If you would like to contact me, I may be reached on Facebook, through email at acr02011@mymail.pomona.edu, or through my phone at (314) 440-8830.

## A.2 D Resources

### A.2.1 Installation

D has a number of compiler implementations, but we will be using dmd, the reference compiler.

On Windows or Macs, you may install D by using the binary installer.

On Linux, you can likely install D with your distro's package manager, but it might not be in the official repositories. For example, on Ubuntu, you will have to add the d-apt repository and install dmd.

```
$ sudo wget \
    http://master.dl.sourceforge.net/project/d-apt/files/d-apt.list \
    -O /etc/apt/sources.list.d/d-apt.list
$ sudo apt-get update
$ sudo apt-get -y --allow-unauthenticated install \
    --reinstall d-apt-keyring && sudo apt-get update
$ sudo apt-get install dmd
```

If you're using project, you'll have to build D from source. I've written a shell script to do this, as the process is pretty involved.

### A.2.2 D Documents

- D Reference
- Programming in D

You should read the following resources before getting started with D, though you will likely find other sections helpful:

- All Chapters up to Strings
- Lifetimes
- Member Functions
- Pointers
- Templates
- Parallelism
- Files

## A.3 Rust Resources

### A.3.1 Installation

Rust is a very volatile language, with breaking changes being brought into the master branch almost every day. Most developers who use Rust use the nightly releases which means that they must constantly update their code to deal with the most recent breaking changes. Since I do not want to introduce any additional overhead to the project, we will be sticking to the **1.0.0-alpha.2** release of Rust. Please ensure that you use this version in order to keep source compatibility during your time working on the project.

You may install Rust using one of the 1.0.0-alpha.2 (**not nightly**) binary installers located here.

If you have problems with the binary installers, you may install Rust from source in the following way. If you are attempting to build Rust on project, you may skip step 1, as the dependencies are already installed.

1. Install dependencies

   Use homebrew or your distro's package manager to install

   - g++ 4.7 or clang++ 3.7
   - python 2.6 or later (not 3.x)
   - GNU make 3.81 or later
   - curl
   - git

2. Download and build Rust:

   ```
   $ git clone https://github.com/rust-lang/rust.git && cd rust
   $ git checkout 1.0.0-alpha.2
   $ ./configure --prefix=$PWD
   $ make -j5 && make install
   ```

   This will take a while, on my laptop it took just over an hour to compile.

3. Add Rust to your path.

   ```
   echo "export PATH=\$PATH:$PWD/bin" >> ~/.bashrc
   source ~/.bashrc
   ```

   Test that rust installed correctly.

   ```
   $ rustc --version
   rustc 1.0.0-dev
   ```

### A.3.2 Documentation

Note that he docs are pinned to the 1.0.0-alpha.2 release.

- Rust Documentation
- Rust by Example

*Note:* since Rust is changing so rapidly, there is a chance that Rust by Example will be outdated. The official documentation and associated book will be correct, however.

You should read the following resources before getting starting with Rust, though you will likely find other sections helpful:

- All of Chapter 2: Basics (skip the installation chapter)
- Method Syntax (classes)
- Pointers and Ownership
- Generics
- Concurrency
- File I/O

# Appendix B

# Solutions

This appendix includes my solutions to the problems that were given to my volunteers. The code in each example strives to be as idiomatic as possible in each language.

## B.1 Hello, World

### B.1.1 C++

```cpp
#include <iostream>

int main() {
    std::cout << "Hello world!" << std::endl;
}
```

### B.1.2 D

```d
import std.stdio;

void main() {
    writeln("Hello world!");
}
```

### B.1.3 Rust

```rust
fn main() {
    println!("Hello world!");
}
```

## B.2 Sentence Splitter

### B.2.1 C++

```cpp
#include <cctype>
#include <iostream>
#include <iterator>
#include <set>
#include <vector>

void splitSentences(std::string input) {
    if (input.length() < 3) {
        // Just assume that it's a single sentence.
        std::cout << input << std::endl;
    }

    size_t pos = 0;                     // The period we are considering.
    size_t sentence_boundary = 0;       // The start of the current sentence.

    while ((pos = input.find_first_of(".?!", pos + 1)) != std::string::npos) {
        // Periods followed by whitespace followed by a lower case letter are
```

```
        // not sentence boundaries.
        if (pos + 2 < input.length() &&
                std::isspace(input.at(pos + 1)) &&
                std::islower(input.at(pos + 2))) {
            continue;
        }

        // Periods followed by a digit with no intervening whitespace are not
        // sentence boundaries.
        if (pos + 1 < input.length() &&
                std::isdigit(input.at(pos + 1))) {
            continue;
        }

        // Periods followed by whitespace and then an upper case letter, but
        // preceded by any of a short list of titles are not sentence
        // boundaries.
        const std::set<std::string> titles = {"Mr.", "Mrs.", "Dr.", "Jr."};
        if ((pos >= 2 && titles.count(input.substr(pos - 2, 3))) ||
                (pos >= 3 && titles.count(input.substr(pos - 3, 4)))) {
            continue;
        }

        // Periods internal to a sentence of letters with no adjacent
        // whitespace are not sentence boundaries (e.g., websites).
        if (pos + 1 < input.length() &&
                std::isalpha(input.at(pos - 1)) &&
                std::isalpha(input.at(pos + 1))) {
            continue;
        }

        // Periods followed by certain kinds of punctuation (notably comma and
        // more periods) are probably not sentence boundaries.
        if (pos + 1 < input.length() && input.at(pos + 1) == ',') {
            continue;
        }

        // Consume any periods that are next to each other.
        while (pos + 1 < input.length() && input.at(pos + 1) == '.') {
            pos++;
        }

        // We passed all the special conditions, so we probably have a
        // sentence.
        std::string sentence =
            input.substr(sentence_boundary, pos - sentence_boundary + 1);
        std::cout << sentence << std::endl;

        // Start the next sentence after the period and the initial space.
        sentence_boundary = pos + 2;
    }
}

int main() {
    std::istream_iterator<char> end_of_stream;
```

```
    std::istream_iterator<char> input_it (std::cin >> std::noskipws);

    std::string input(input_it, end_of_stream);
    splitSentences(input);
}
```

### B.2.2 D

```
import std.algorithm;
import std.ascii : isDigit;
import std.container;
import std.file;
import std.range;
import std.stdio;
import std.uni : isAlpha, isLower, isSpace;

void splitSentences(string input) {
    if (input.length < 3) {
        // Assume that the input is a single sentence.
        writeln(input);
    }

    ptrdiff_t pos = 0;                  // The period we are considering.
    ptrdiff_t sentence_boundary = 0;    // The start of the current sentence.

    while (canFind(input[pos + 1 .. $], ".", "?", "!")) {
        pos += countUntil(input[pos + 1 .. $], ".", "?", "!") + 1;

        if (pos + 2 < input.length &&
                isSpace(input[pos + 1]) &&
                isLower(input[pos + 2])) {
            continue;
        }

        if (pos + 1 < input.length && isDigit(input[pos + 1])) {
            continue;
        }

        // Create a est of allowed titles
        auto titles = redBlackTree("Mr.", "Mrs.", "Dr.", "Jr.");
        if ((pos >= 2 && input[pos - 2 .. pos + 1] in titles) ||
                (pos >= 3 && input[pos - 3 .. pos + 1] in titles)) {
            continue;
        }

        if (pos + 1 < input.length &&
                isAlpha(input[pos - 1]) && isAlpha(input[pos + 1])) {
            continue;
        }

        if (pos + 1 < input.length && input[pos + 1] == ',') {
            continue;
        }
```

```
        while (pos + 1 < input.length && input[pos + 1] == '.') {
            pos++;
        }

        string sentence =
                input[sentence_boundary .. pos + 1];
        writeln(sentence);

        sentence_boundary = pos + 2;
    }
}

void main() {
    string input = join(stdin.byLineCopy().array());
    splitSentences(input);
}
```

### B.2.3 Rust

```
#![feature(old_io,unicode)]

use std::old_io as io;
use std::collections::HashSet;

fn split_sentences(input: String) {
    if input.len() < 3 {
        // Assume that the input is a single sentence.
        println!("{}", input);
    }

    let chars: Vec<char> = input.chars().collect();
    let mut sentence_boundary = 0;

    // Find the next plausible sentence boundary.
    for (mut pos, _) in chars.iter().enumerate().filter(
            |&(_, &c)| c == '.' || c == '!' || c == '?') {
        // Periods followed by whitespace followed by a lower case letter are
        // not sentence boundaries.
        if pos + 2 < input.len() &&
              chars[pos + 1].is_whitespace() &&
              chars[pos + 2].is_lowercase() {
            continue
        }

        // Periods followed by a digit with no intervening whitespaces are not
        // sentence boundaries.
        if pos + 1 < input.len() && chars[pos + 1].is_digit(10) {
            continue
        }

        // Periods folled by whitespace and then an upper case letter, but
        // preceded by any of a short list of titles are not sentence
        // boundaries.
        let titles: HashSet<&'static str> =
```

```
                ["Mr.", "Mrs.", "Dr.", "Jr."].iter().map(|&x| x).collect();
        if pos >= 2 {
            let two_previous_chars: String =
                chars[pos-2..pos+1].iter().map(|&x| x).collect();
            if titles.contains(&two_previous_chars[..]) {
                continue
            }
        }
        if pos >= 3 {
            let three_previous_chars: String =
                chars[pos-3..pos+1].iter().map(|&x| x).collect();
            if titles.contains(&three_previous_chars[..]) {
                continue
            }
        }

        // Periods internal to a sentence of letter with no adjacent whitespace
        // are not sentence boundaries (e.g., websites).
        if pos + 1 < input.len() &&
              chars[pos + 1].is_alphabetic() &&
              chars[pos - 1].is_alphabetic() {
            continue
        }

        if pos + 1 < input.len() && chars[pos + 1] == ',' {
            continue
        }

        // Do nothing, we are just trying to consume any periods that are next
        // to each other.
        while pos + 1 < input.len() && chars[pos + 1] == '.' {
            pos += 1;
        }

        // We passed all the special conditions, so we probably have a sentence.
        if sentence_boundary < pos {
            let sentence: String =
                    chars[sentence_boundary..pos + 1].iter()
                    .map(|&x| x).collect();
            println!("{}", sentence);
        }

        sentence_boundary = pos + 2;
    }
}

fn main() {
    let mut stdin = io::stdin();
    let input = String::from_utf8(stdin.read_to_end().unwrap()).unwrap();
    split_sentences(input);
}
```

## B.3 Integer Linked List

### B.3.1 C++

```cpp
#ifndef INT_LINKED_LIST_HPP
#define INT_LINKED_LIST_HPP

#include <memory>

class IntLinkedList {
    private:
        struct Node {
            int element;
            std::shared_ptr<Node> next;

            Node(int element);
            Node(int element, std::shared_ptr<Node> next);
        };

        std::shared_ptr<Node> headNode = nullptr;
        size_t num_nodes;

        std::shared_ptr<Node> traverse(size_t num_elements) const;
        void ensure_capacity(size_t index) const;
    public:
        int get(size_t index) const;
        void insert(size_t index, int element);
        int remove(size_t index);
        int head() const;
        int tail() const;
        size_t size() const;
};

#endif  // INT_LINKED_LIST_H
```

```cpp
#include <memory>
#include <stdexcept>

#include "int_linked_list.hpp"

IntLinkedList::Node::Node(int element) : element(element), next(nullptr) {}
IntLinkedList::Node::Node(int element, std::shared_ptr<Node> next) :
    element(element), next(next) {}

std::shared_ptr<IntLinkedList::IntLinkedList::Node>
IntLinkedList::traverse(size_t num_elements) const {
    std::shared_ptr<Node> current (headNode);

    size_t elements_to_go = num_elements;
    if (num_elements > this->size()) {
        throw std::out_of_range("index out of range");
    }
```

```
    while (current->next != nullptr) {
        --elements_to_go;
        current = current->next;
    }

    return current;
}

void IntLinkedList::insert(size_t index, int element) {
    auto current = traverse(index);
    current->next = std::make_shared<Node> (element);
    ++num_nodes;
}

int IntLinkedList::remove(size_t index) {
    // Get the element just before
    auto current = traverse(index - 1);

    int value = current->next->element;
    current->next = current->next->next;

    return value;
}

size_t IntLinkedList::size() const {
    return this->num_nodes;
}

int IntLinkedList::get(size_t index) const {
    return traverse(index)->element;
}

int IntLinkedList::head() const {
    return headNode->element;
}

int IntLinkedList::tail() const {
    return traverse(num_nodes - 1)->element;
}
```

### B.3.2 D

```
module intlist;

import std.string : format;
import std.stdio;

class IntegerLinkedList {
    private Node headNode = null;
    private size_t numNodes = 0;

    private class Node {
        int element;
        Node next = null;
```

```
    this(int element) {
        this.element = element;
    }

    this(int element, Node next) {
        this(element);
        this.next = next;
    }
}

private void ensureCapacity(size_t index) const {
    if (index > this.size()) {
        throw new Exception(
                format("index %d out of range", index));
    }
}

// Returns a pointer to the Node represented by the given index.
private Node* traverse(size_t index) const {
    ensureCapacity(index);

    size_t elementsToGo = index;

    auto current = cast(Node*)&headNode;

    for (int i = 0; i < elementsToGo; i++) {
        current = &current.next;
    }

    return current;
}

void insert(size_t index, int element) {
    ensureCapacity(index);

    auto current = traverse(index);
    if (current !is null) {
        *current = new Node(element, *current);
    } else {
        *current = new Node(element);
    }
    ++numNodes;
}

int remove(size_t index) {
    ensureCapacity(index);

    auto current = traverse(index);
    int value = current.element;
    *current = current.next;
    numNodes--;
    return value;
}
```

```
    int head() const {
        if (size() == 0) {
            throw new Exception("the list is empty");
        }

        return headNode.element;
    }

    int tail() const {
        if (size() == 0) {
            throw new Exception("the list is empty");
        }

        return traverse(numNodes - 1).element;
    }

    int get(size_t index) const {
        return traverse(index).element;
    }

    size_t size() const {
        return numNodes;
    }
}
```

### Test Case

```
module intlisttest;

import intlist;

/// Lists have no elements when they are created.
unittest {
    auto list = new IntegerLinkedList;
    assert(list.size() == 0);
}

// Add a single element
unittest {
    auto list = new IntegerLinkedList;
    list.insert(0, 0);
    assert(list.size() == 1);
    assert(list.head() == 0);
}

/// Remove a single element
unittest {
    auto list = new IntegerLinkedList;
    list.insert(0, 47);
    assert(list.size() == 1);
    assert(list.remove(0) == 47);
    assert(list.size() == 0);
}
```

```
/// Retrieve a single element
unittest {
    auto list = new IntegerLinkedList;
    list.insert(0, 42);
    assert(list.get(0) == 42);
    assert(list.size() == 1);
}

/// Add multiple elements
unittest {
    auto list = new IntegerLinkedList;
    list.insert(0, 1);
    list.insert(1, 2);
    assert(list.size() == 2);
    assert(list.head() == 1);
}

/// Remove multiple elements
unittest {
    auto list = new IntegerLinkedList;
    list.insert(0, 1);
    list.insert(1, 2);
    assert(list.remove(1) == 2);
    assert(list.remove(0) == 1);
}

/// Test that head and tail work as expected
unittest {
    auto list = new IntegerLinkedList;
    list.insert(0, 2);
    list.insert(0, 1);
    list.insert(2, 3);
    assert(list.head() == 1);
    assert(list.tail() == 3);
}

void main() {}
```

### B.3.3 Rust

```
use std::mem;

pub struct IntegerLinkedList {
    head: Option<Node>,
    num_nodes: usize,
}

struct Node {
    next: Option<Box<Node>>,
    element: i32
}

impl IntegerLinkedList {
    pub fn new() -> IntegerLinkedList {
```

```
    IntegerLinkedList { head: None, num_nodes: 0 }
}

pub fn size(&self) -> usize {
    self.num_nodes
}

pub fn insert(&mut self, index: usize, element: i32) {
    if index == 0 && self.num_nodes > 0 {
        let old_head = mem::replace(&mut self.head, None);
        let new_head =
            Node::new_with_tail(element, Some(Box::new(old_head.unwrap())));
        self.head = Some(new_head);
        self.num_nodes += 1;
        return
    }

    match self.head {
        Some(ref mut head) => {
            head.insert(index, element);
        },
        None => {
            if index == 0 {
                self.head = Some(Node::new(element));
            } else {
                panic!("list index out of bounds")
            }
        }
    }
    self.num_nodes += 1
}

pub fn remove(&mut self, index: usize) -> i32 {
    let val;
    match self.head {
        Some(ref mut head) => {
            val = head.remove(index);
        },
        None => panic!("list index out of bounds")
    }
    self.num_nodes -= 1;
    val
}

pub fn head(&self) -> i32 {
    match self.head {
        Some(ref head) => {
            head.element
        },
        None => panic!("no head of list")
    }
}

pub fn tail(&self) -> i32 {
    self.get(self.num_nodes - 1)
```

```
    }

    pub fn get(&self, index: usize) -> i32 {
        match self.head {
            Some(ref head) => {
                head.get(index)
            },
            None => panic!("list index out of bounds")
        }
    }
}

impl Node {
    pub fn new(element: i32) -> Node {
        Node { next: None, element: element }
    }

    pub fn new_with_tail(element: i32, next: Option<Box<Node>>) -> Node {
        Node { next: next, element: element }
    }

    pub fn insert(&mut self, index: usize, element: i32) {
        match self.next {
            Some(ref mut next) => {
                if index > 0 {
                    return next.insert(index - 1, element);
                }
            },
            None => {}
        };

        let old_node = mem::replace(&mut self.next, None);
        let mut new_node = Box::new(Node::new(element));
        new_node.next = old_node;
        self.next = Some(new_node)
    }

    pub fn remove(&mut self, index: usize) -> i32 {
        let ret;

        match self.next {
            Some(ref mut next) => {
                if index > 0 {
                    return next.remove(index - 1)
                } else {
                    ret = self.element
                }
            }
            None => if index == 0 {
                ret = self.element
            } else {
                panic!("list index out of bounds")
            }
        };
```

```
        self.next = match self.next {
            Some(ref mut next) => { mem::replace(&mut next.next, None) },
            None => None
        };

        ret
    }

    pub fn get(&self, index: usize) -> i32 {
        if index == 0 {
            self.element
        } else {
            match self.next {
                Some(ref next) => next.get(index - 1),
                None => panic!("list index out of bounds")
            }
        }
    }
}
```

### Test Case

```
extern crate int_linked_list;

use int_linked_list::IntegerLinkedList;

// Empty list tests
#[test]
fn test_empty() {
    let list = IntegerLinkedList::new();
    assert_eq!(list.size(), 0);
}

#[test]
fn test_add_empty() {
    let mut list = IntegerLinkedList::new();
    list.insert(0, 2);
    assert_eq!(list.size(), 1);
    assert_eq!(list.head(), 2);
    assert_eq!(list.tail(), 2);
}

// Single element tests
#[test]
fn test_remove_one() {
    let mut list = IntegerLinkedList::new();
    list.insert(0, 47);
    assert_eq!(list.size(), 1);
    assert_eq!(list.remove(0), 47);
    assert_eq!(list.size(), 0);
}

#[test]
fn test_get_one() {
```

```rust
    let mut list = IntegerLinkedList::new();
    list.insert(0, 42);
    assert_eq!(list.get(0), 42);
    assert_eq!(list.size(), 1);
}

// Adding multiple elements
#[test]
fn test_add_two() {
    let mut list = IntegerLinkedList::new();
    list.insert(0, 1);
    list.insert(1, 2);
    assert_eq!(list.size(), 2);
    assert_eq!(list.head(), 1);
}

#[test]
fn test_remove_two() {
    let mut list = IntegerLinkedList::new();
    list.insert(0, 1);
    list.insert(1, 2);
    assert_eq!(list.remove(1), 2);
    assert_eq!(list.remove(0), 1);
}

#[test]
fn test_head_tail() {
    let mut list = IntegerLinkedList::new();
    list.insert(0, 2);
    list.insert(0, 1);
    list.insert(2, 3);
    assert_eq!(list.head(), 1);
    assert_eq!(list.tail(), 3);
}
```

## B.4 Generic Array List

### B.4.1 C++

```cpp
#ifndef GENERIC_ARRAY_LIST_HPP
#define GENERIC_ARRAY_LIST_HPP

#include <memory>

template <typename E>
class GenericArrayList {
    private:
        const size_t initialSize = 10;
        std::unique_ptr<E[]> backingArray;
        size_t backingArraySize = initialSize;
        void resize(size_t newSize);
        size_t numElements;
    public:
```

```
        GenericArrayList();
        void insert(size_t index, E element);
        E remove(size_t index);
        size_t size() const;
        E head() const;
        E tail() const;
        E get(size_t index) const;
};


#endif  // GENERIC_ARRAY_LIST_HPP
```

```
#include "generic-array-list.hpp"

#include <iterator>

template <typename E>
GenericArrayList<E>::GenericArrayList() : backingArray(new E[initialSize]) {}

template<typename E>
void GenericArrayList<E>::resize(size_t newSize) {
    std::unique_ptr<E[]> newArray = new E[newSize];

    std::copy(std::begin(backingArray), std::end(backingArray),
            std::begin(newArray));

    backingArray = newArray;
}

template<typename E>
void GenericArrayList<E>::insert(size_t index, E element) {
    if (size() + 1 > backingArraySize) {
        resize(backingArraySize * 2);
    }

    // Shift the elements up
    for (size_t i = size(); i > index; --i) {
        backingArray[i] = backingArray[i - 1];
    }

    backingArray[index] = element;

    numElements++;
}

template<typename E>
E GenericArrayList<E>::remove(size_t index) {
    E value = backingArray[index];

    // Shift the elements down
    for (size_t i = index; i < size() - 1; ++i) {
        backingArray[i] = backingArray[i + 1];
    }
```

```
    numElements--;

    return value;
}

template<typename E>
size_t GenericArrayList<E>::size() const {
    return numElements;
}

template<typename E>
E GenericArrayList<E>::head() const {
    return backingArray[0];
}

template<typename E>
E GenericArrayList<E>::tail() const {
    return backingArray[numElements];
}
```

### B.4.2 D

```
module genericarraylist;

import std.algorithm : copy;

class ArrayList(T) {
    private T[] backingArray;
    private size_t theSize;

    this() {
        const auto initialSize = 10;
        backingArray = new T[10];
    }

    private void resize(size_t newSize) {
        T[] newArray = new T[newSize];
        copy(newArray, backingArray);
        backingArray = newArray;
    }

    void insert(size_t index, T element) {
        if (size() + 1 > backingArray.length) {
            resize(backingArray.length * 2);
        }

        copy(backingArray[index + 1 .. $], backingArray[index .. $ - 1]);
        backingArray[index] = element;

        theSize++;
    }

    T remove(size_t index) {
        T value = backingArray[index];
```

```
        copy(backingArray[index .. $-1], backingArray[index + 1.. $]);

        theSize--;

        return value;
    }

    size_t size() {
        return theSize;
    }

    T head() {
        return backingArray[0];
    }

    T tail() {
        return backingArray[theSize - 1];
    }

    T get(size_t index) {
        return backingArray[index];
    }
}
```

### Test Case

```
module genericarraylisttest;

import genericarraylist;

/// Lists have no elements when they are created.
unittest {
    auto list = new ArrayList!(int);
    assert(list.size() == 0);
}

/// Add a single element
unittest {
    auto list = new ArrayList!(int);
    list.insert(1, 0);
    assert(list.size() == 1);
    assert(list.head() == 0);
}

/// Remove a single element
unittest {
    auto list = new ArrayList!(int);
    list.insert(0, 47);
    assert(list.size() == 1);
    assert(list.remove(0) == 47);
    assert(list.size() == 0);
}
```

```d
/// Retrieve a single element
unittest {
    auto list = new ArrayList!(int);
    list.insert(0, 42);
    assert(list.get(0) == 42);
    assert(list.size() == 1);
}

/// Add multiple elements
unittest {
    auto list = new ArrayList!(int);
    list.insert(0, 1);
    list.insert(1, 2);
    assert(list.size() == 2);
    assert(list.head() == 1);
}

/// Remove multiple elements
unittest {
    auto list = new ArrayList!(int);
    list.insert(0, 1);
    list.insert(1, 2);
    assert(list.remove(1) == 2);
    assert(list.remove(0) == 1);
}

void main() { }
```

### B.4.3 Rust

```rust
pub struct ArrayList<T> {
    size: usize,
    backing_array: Vec<T>
}

impl<T: Clone> ArrayList<T> {
    pub fn new() -> ArrayList<T> {
        ArrayList { size: 0, backing_array: Vec::with_capacity(10) }
    }

    pub fn size(&self) -> usize {
        self.size
    }

    pub fn head(&self) -> Option<T> {
        match self.size > 0 {
            true => Some(self.backing_array[0].clone()),
            false => None
        }
    }

    pub fn insert(&mut self, index: usize, element: T) {
        self.size += 1;
        self.backing_array.insert(index, element)
```

```
    }

    pub fn remove(&mut self, index: usize) -> T {
        self.size -= 1;
        self.backing_array.remove(index)
    }

    pub fn get(&self, index: usize) -> T {
        self.backing_array[index].clone()
    }
}
```

### Test Case

```
extern crate generic_array_list;

use generic_array_list::ArrayList;

// Empty list tests
#[test]
fn test_empty() {
    let list: ArrayList<i32> = ArrayList::new();
    assert_eq!(list.head(), None);
    assert_eq!(list.size(), 0);
}

#[test]
fn test_add_empty() {
    let mut list: ArrayList<i32> = ArrayList::new();
    list.insert(0, 2);
    assert_eq!(list.size(), 1);
    assert_eq!(list.head(), Some(2));
}

// Single element tests
#[test]
fn test_remove_one() {
    let mut list: ArrayList<i32> = ArrayList::new();
    list.insert(0, 47);
    assert_eq!(list.size(), 1);
    assert_eq!(list.remove(0), 47);
    assert_eq!(list.size(), 0);
}

#[test]
fn test_get_one() {
    let mut list: ArrayList<i32> = ArrayList::new();
    list.insert(0, 42);
    assert_eq!(list.get(0), 42);
    assert_eq!(list.size(), 1);
}

// Adding multiple elements
#[test]
```

```rust
fn test_add_two() {
    let mut list: ArrayList<i32> = ArrayList::new();
    list.insert(0, 1);
    list.insert(1, 2);
    assert_eq!(list.size(), 2);
    assert_eq!(list.head(), Some(1));
}

#[test]
fn test_remove_two() {
    let mut list: ArrayList<i32> = ArrayList::new();
    list.insert(0, 1);
    list.insert(1, 2);
    assert_eq!(list.remove(1), 2);
    assert_eq!(list.remove(0), 1);
}
```

## B.5 Parallel Merge Sort

### B.5.1 C++

```cpp
#include <future>
#include <iostream>
#include <string>
#include <vector>

// Merge two sorted arrays into a single sorted array.
template <typename E>
std::vector<E> merge(std::vector<E>& left, std::vector<E>& right) {
    std::vector<E> result;

    while(!left.empty() && !right.empty()) {
        if (*left.begin() <= *right.begin()) {
            result.push_back(*left.begin());
            left.erase(left.begin());;
        } else {
            result.push_back(*right.begin());
            right.erase(right.begin());
        }
    }

    // If there are any remaining elements, then push them on to the result
    // vector.
    for (auto&& element : left) {
        result.push_back(element);
    }

    for (auto&& element : right) {
        result.push_back(element);
    }

    return result;
}
```

```cpp
// Perform a recursive mergesort.
template <typename E>
std::vector<E>  mergesort(const std::vector<E>& elements) {
    if (elements.size() <= 1) {
        return elements;
    }

    size_t middle = elements.size() / 2;
    std::vector<E> left(elements.begin(), elements.begin() + middle);
    std::vector<E> right(elements.begin() + middle, elements.end());

    // If we have fewer than 10 elements, don't spawn a new thread.
    if (elements.size() < 10) {
        left = mergesort(left);
        right = mergesort(right);
    } else {
        // Spawn a new task sorting the left side, and perform the right-side
        // merge on the same thread.
        auto leftMerge = std::async(std::launch::async, mergesort<E>, left);
        right = mergesort(right);

        // Wait for the left side to finish.
        left = leftMerge.get();
    }

    return merge(left, right);
}

int main() {
    std::vector<int> elements;

    // Parse integers from stdin.
    for (std::string line; std::getline(std::cin, line);) {
        elements.push_back(std::stoi(line));
    }

    // Perform the sort, and print sorted elements to stdout.
    elements = mergesort(elements);
    for (auto&& element : elements) {
        std::cout << element << std::endl;
    }
}
```

### B.5.2 D

```d
import std.algorithm;
import std.array;
import std.conv;
import std.file;
import std.parallelism : task;
import std.stdio;

T[] merge(T)(T[] left, T[] right) {
```

```
    T[] result;

    while (left.length && right.length) {
        if (left[0] < right[0]) {
            result ~= left[0];
            left = remove(left, 0);
        } else {
            result ~= right[0];
            right = remove(right, 0);
        }
    }

    foreach (ref element; left) {
        result ~= element;
    }

    foreach (ref element; right) {
        result ~= element;
    }

    return result;
}

T[] mergesort(T)(T[] elements) {
    if (elements.length <= 1) {
        return elements;
    }

    T[] left = elements[0 .. $ / 2];
    T[] right = elements[$ / 2 .. $];

    if (elements.length < 10) {
        left = mergesort(left);
        right = mergesort(right);
    } else {
        auto leftMergeTask = task!mergesort(left);
        leftMergeTask.executeInNewThread();
        right = mergesort(right);
        left = leftMergeTask.yieldForce();
    }

    return merge(left, right);
}

void main() {
    auto elements = stdin
                        .byLine()
                        .map!(to!int)
                        .array();

    elements = mergesort(elements);
    foreach (ref element; elements) {
        writeln(element);
    }
}
```

### B.5.3 Rust

```rust
#![feature(old_io)]
#![feature(collections)]

use std::old_io as io;
use std::thread;

fn merge<E>(left: &mut Vec<E>, right: &mut Vec<E>) -> Vec<E>
        where E: PartialOrd + Clone {
    let mut result = Vec::new();

    while !left.is_empty() && !right.is_empty() {
        if left[0] < right[0] {
            result.push(left.remove(0));
        } else {
            result.push(right.remove(0));
        }
    }

    result.push_all(&left);
    result.push_all(&right);

    return result;
}

fn mergesort<E>(elements: Vec<E>) -> Vec<E>
        where E: PartialOrd + Send + Sync + Clone {
    if elements.len() <= 1 {
        return elements;
    }

    let (left_slice, right_slice) = elements.split_at(elements.len() / 2);

    let mut left = left_slice.to_vec();
    let mut right = right_slice.to_vec();

    if elements.len() < 10 {
        left = mergesort(left);
        right = mergesort(right);
    } else {
        let merge_left = thread::scoped(move || { mergesort(left) });
        right = mergesort(right);

        left = merge_left.join();
    }

    return merge(&mut left, &mut right);
}

fn main() {
    let mut stdin = io::stdin();

    let elements = stdin.lock().lines().map(|result| {
        match result {
```

```
            Ok(string) => match string.trim().parse::<i32>() {
                Ok(number) => number,
                Err(e) => panic!(e)
            },
            Err(e) => panic!(e)
        }
    }).collect();

    for element in mergesort(elements) {
        println!("{}", element)
    }
}
```

## B.6 Brainfuck Interpreter

### B.6.1 C++

```
#include <array>
#include <cstdio>
#include <fstream>
#include <iostream>
#include <iterator>
#include <map>
#include <stack>

// The number of cells to use in memory.
const int numCells = 30000;

using cell_ptr = std::array<int, numCells>::iterator;

/*
 * Returns a map that maps where brackets should jump to.
 *
 * That is, a program containing a '[' at character 4 and a ']' at character 8
 * would return a map containing (4, 8) and (8, 4).
 */
std::map<int, int> parseLabels(std::ifstream &file) {
    std::map<int, int> labels;
    std::stack<int> parsedLabels;

    char c;
    while(file >> c) {
        if (c == '[') {
            parsedLabels.push(file.tellg());
        } else if (c == ']') {
            if (parsedLabels.empty()) {
                std::cerr <<
                    "Error: ']' encountered without matching ']'." <<
                    std::endl;
                std::exit(EXIT_FAILURE);
            }

            int lastLabel = parsedLabels.top();
```

```
            parsedLabels.pop();

            labels.insert(std::make_pair(file.tellg(), lastLabel));
            labels.insert(std::make_pair(lastLabel, file.tellg()));
        }
    }

    // We're done reading the file, so we need to set it back to the beginning.
    file.clear();
    file.seekg(std::fstream::beg);

    return labels;
}

/*
 * Runs one step of the interpreter by reading the current character the file
 * pointer is pointing at and modifies the program state accordingly.
 */
void processOpCode(std::ifstream &file, std::map<int, int> &labels,
        std::array<int, numCells> &cells, cell_ptr &pointer) {
    char opCode = file.get();

    switch (opCode) {
        case '>':
            ++pointer;
            break;
        case '<':
            --pointer;
            break;
        case '+':
            ++*pointer;
            break;
        case '-':
            --*pointer;
            break;
        case '.':
            std::putchar(*pointer);
            break;
        case ',':
            *pointer = std::getchar();
            break;
        case '[':
            if (!*pointer) file.seekg(labels.at(file.tellg()));
            break;
        case ']':
            if (*pointer) file.seekg(labels.at(file.tellg()));
            break;
    }
}

int main(int argc, char** argv) {
    if (argc < 2) {
        std::cout << "Usage: brainfsck [.bf file]" << std::endl;
        return EXIT_FAILURE;
    }
```

```
    std::string fileName(argv[1]);
    std::ifstream file(fileName, std::ifstream::in);

    std::array<int, numCells> cells = {};
    cell_ptr pointer = cells.begin();
    std::map<int, int> labels = parseLabels(file);

    while (file.good()) {
        processOpCode(file, labels, cells, pointer);
    }

    return EXIT_SUCCESS;
}
```

### B.6.2 D

```
import std.algorithm;
import std.c.stdlib;
import std.range;
import std.stdio;

const int numCells = 30000;

ulong[ulong] parseLabels(File file) {
    ulong[ulong] labels;
    ulong[] parsedLabels;

    auto buf = new ubyte[](cast(size_t)file.size); file.rawRead(buf);

    foreach (ulong i, char c; buf.enumerate(1)) {
        if (c == '[') {
            parsedLabels ~= i;
        } else if (c == ']') {
            if (parsedLabels.empty) {
                stderr.writefln(
                        "Error: ']' encountered without matching ']'.");
                exit(1);
            }

            auto lastLabel = parsedLabels.back;
            parsedLabels.popBack();

            labels[i] = lastLabel;
            labels[lastLabel] = i;
        }
    }

    file.rewind();

    return labels;
}

void processOpCode(File file, ulong[ulong] labels, ref int[numCells] cells,
```

```
                    ref int cellIndex) {
    char opCode;
    file.readf("%c", &opCode);

    switch (opCode) {
        case '>':
            ++cellIndex;
            break;
        case '<':
            --cellIndex;
            break;
        case '+':
            ++cells[cellIndex];
            break;
        case '-':
            --cells[cellIndex];
            break;
        case '.':
            write(cast(char)cells[cellIndex]);
            break;
        case ',':
            stdin.readf("%c", &cells[cellIndex]);
            break;
        case '[':
            if (!cells[cellIndex]) {
                file.seek(labels[file.tell()]);
            }
            break;
        case ']':
            if (cells[cellIndex]) {
                file.seek(labels[file.tell()]);
            }
            break;
        default:
    }
}

void main(string[] args) {
    if (args.length < 2) {
        writeln("Usage: brainfsck [.bf file]");
        exit(1);
    }

    auto fileName = args[1];
    auto file = File(fileName, "r");

    int[numCells] cells;
    auto cellPointer = 0;
    auto labels = parseLabels(file);

    while (!file.eof()) {
        processOpCode(file, labels, cells, cellPointer);
    }
}
```

### B.6.3 Rust

```rust
#![feature(env)]
#![feature(old_io)]
#![feature(old_path)]

use std::collections::HashMap;
use std::char;
use std::old_io as io;
use std::old_io::{File, IoResult};
use std::old_io::SeekStyle::*;
use std::env::args;

const NUM_CELLS: usize = 30000;


fn parse_labels(file: &mut File) -> IoResult<HashMap<u64, u64>> {
    let mut labels: HashMap<u64, u64> = HashMap::new();
    let mut parsed_labels = Vec::new();

    let string = try!(file.read_to_string());

    for (i, c) in string.chars().enumerate() {
        let pos = i + 1;

        match c {
            '[' => {
                parsed_labels.push(pos);
            },
            ']' => {
                if parsed_labels.is_empty() {
                    panic!("']' encountered without matching ']'");
                }

                let last_label = parsed_labels.pop().unwrap();
                labels.insert(pos as u64, last_label as u64);
                labels.insert(last_label as u64, pos as u64);
            }
            _ => {}
        }
    }

    try!(file.seek(0, SeekSet));

    Ok(labels)
}

fn seek_to_label_match(file: &mut File, labels: &HashMap<u64, u64>)
        -> IoResult<()> {
    let file_pos = &try!(file.tell());
    let label_match = *labels.get(file_pos).unwrap() as i64;
    try!(file.seek(label_match, SeekSet));

    Ok(())
}
```

```
fn process_op_code(file: &mut File, labels: &HashMap<u64, u64>,
                   cells: &mut [i32], index: &mut usize) -> IoResult<()> {
    let op_code = char::from_u32(try!(file.read_byte()) as u32).unwrap();

    match op_code {
        '>' => *index += 1,
        '<' => *index -= 1,
        '+' => cells[*index] += 1,
        '-' => cells[*index] -= 1,
        '.' => print!("{}", char::from_u32(cells[*index] as u32).unwrap()),
        ',' => {
            let mut stdin = io::stdin();
            cells[*index] = try!(stdin.lock().read_byte()) as i32;
        },
        '[' => if cells[*index] == 0 {
            let _ = seek_to_label_match(file, labels);
        },
        ']' => if cells[*index] != 0 {
            let _ = seek_to_label_match(file, labels);
        },
        _ => {}
    }

    Ok(())
}

fn main() {
    let args: Vec<_> = args().collect();

    if args.len() < 2 {
        println!("Usage: brainfsck [.bf file]");
        return
    }

    let file_path = Path::new(&args[1]);
    let mut file = match File::open(&file_path) {
        Ok(file) => file,
        Err(err) => panic!(err),
    };

    let mut cells = [0; NUM_CELLS];
    let mut index = 0;
    let labels = match parse_labels(&mut file) {
        Ok(labels) => labels,
        Err(err) => panic!(err)
    };

    while !file.eof() {
        match process_op_code(&mut file, &labels, &mut cells, &mut index) {
            Ok(_) => {}
            Err(err) => {
                // The file can't be read anymore, don't do anything and assume
                // it worked.
                break
```

```
            }
        }
    }
}
```

### B.6.4 Brainfuck Test Case ("Hello World")

```
>++++++++[<++++++++++>-]<.>>+>+>++>[-]+<[>[->+<<++++>]<<]>.+++++++..+++.>
>+++++++.<<<[[-]<[-]>]<+++++++++++++++.>>.+++.------.--------.>>+.>++++.
```