C∀ Collection Library

by

Michael Leslie Brooks

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2026

© Michael Leslie Brooks 2026

Author's Declaration

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

C∀ strives to fix issues in C, chief among them safety. This thesis presents a significant step forward in C∀'s goal to remove unsafe pointer operations. It describes improvements to the C∀ language design to support advanced collection features. These features are implemented across the C∀ compiler and runtime libraries. The results maintain another C∀ goal of offering strong backwards compatibility with C. To achieve these goals, this work leverages preexisting C∀ contributions by prior students, particularly novel applications of the compiler's type system.

Modern programming languages accommodate needs for these three high-level collections (containers): array, linked-list, and string. For all three types, languages and/or their libraries supply varying degrees of high-level mechanisms for manipulating these objects at the bulk and component levels, such as copying, slicing, extracting, and iterating among elements. Unfortunately, typical implementations for the these key types in C cause 60%–70% of the reported software vulnerabilities involving memory errors, where 70%–80% of hacker attack vectors target these types. Therefore, hardening these three C types and suggesting programmers use them as their default types goes a long way to increase memory safety in the majority of C programs.

Specifically, an array is provided that tracks its length internally, relieving the user and implementor from managing explicit length arguments/parameters and stopping buffer-overrun errors. This feature requires augmenting the C∀ type system, making array length available at compile and runtime. A linked-list utility is provided that obviates many user-managed recursive pointers, while catering directly to system-programming using intrusive linking. Finally, a string utility is provided with implicit memory management of text in a specialized heap, removing error-prone buffer management, including overrun, and providing a copy-on-write speed boost. For all three utilities, performance is argued to be on-par or surpass those in other comparable languages. With the array, this case is made by showing complete erasure down to a naked C array, modulo runtime bound checks, which are removable more often than with Java-style length management. With the linked list and string, empirical measures are compared with C and C++ comparable libraries. These collections offer programmers workable alternatives to hand-rolling specialized libraries, which is a huge safety benefit, eliminating many system vulnerabilities. The results establish C∀'s position as a safety-forward programming alternative.

Acknowledgements

My foremost gratitude goes to my wife, Stephanie Brooks. She supported me financially, morally and with gracious patience during this project. Along with our children Abby and Norah, my family went without comforts that it might have enjoyed otherwise, offered a "Dad hug" when most needed, and did everything in between, to make this work possible. Steph continues to inspire me to be more than I am.

My parents, Dr. Robert Brooks and Dr. Bonnie Edwards, served as inspiration to finish, provided financial support, and had my back the whole way. Mum taught me to program when I was in grade school and Dad helped debug my high school programming blunders.

Dr. Peter Buhr, my supervisor, runs a really cool team that makes a really cool product. I cannot imagine better days at work thank those I spend deep inside C∀, with him or with any of our teammates. Peter has created an environment of autonomy and ideation where an unassigned inspiration could become my Chapter 3. I'm grateful that he took me in and never gave up on me.

A surely incomplete attempt to recognize shared successes now proceeds. C∀ teammate Andrew Beach got me through an implementation challenge, at some point, every time I added logic to the C∀ translator. C∀ teammate Dr. Thierry Delisle drew my attention to the linked-list design issue of Section 2.4.7, p. 31 and, correspondingly, the "fake node forms a circle" design option for headed lists of Figure 4.9, p. 89. C∀ teammate Fangren Yu diagnosed the constructor blowup situation of Figure 3.11, p. 65. Surely every C∀ teammate with whom I overlapped shared in design discussions or bug reporting. Among them, Colby Parsons took on the particular challenge of replacing bespoke linked lists in the C∀ concurrency system with mine, which was a valuable source of feedback. Peter provided one jumping-off point for strings, with a C++ implementation of an occasionally compacted byte heap accessed by linked handles. His summer-student Sunjay Varma provided another, in building a first-draft C∀ string API, from which mine evolved.

In supervising this thesis, Peter prompted me to provide many missing elaborations, resulting in probably half of the sentences and diagrams herein. He further saved you, dear reader, from at least double the page count, in stopping my attempts to discuss connections with myriad fascinating topics *not* found herein. My reviewer, Dr. Gregor Richards, managed to point out exactly what was wrong with many parts where I knew I was uncomfortable about something, but couldn't quite figure out why. My reviewer, Dr. Patrick Lam, challenged me to show more relevant context, where I had previously treated a matter with blinders on.

Finally, a special thank-you to Huawei Canada for funding this work.

Table of Contents

Author's Declaration iii

Abstract v

Acknowledgements vii

List of Figures xiii

List of Tables xv

1 Introduction 1

1.1 Array . . . 2
1.2 Linked List . . . 2
1.3 String . . . 3
1.4 Motivation . . . 3
1.5 Which C? . . . 4
1.6 Contributions . . . 5
1.6.1 Array . . . 5
1.6.2 Linked List . . . 5
1.6.3 String . . . 6

2 Background 7

2.1 Ill-Typed Expressions . . . 7
2.2 Reading Declarations . . . 8
2.3 Array . . . 10
2.3.1 Arrays Decay and Pointers Diffract . . . 12
2.3.2 Variable-Length Arrays . . . 16

2.3.3 Multidimensional Arrays . . . 16
2.3.4 Multi-Dimensional Arrays Decay and Pointers Diffract . . . 19
2.3.5 Array Parameter Declaration . . . 20
2.3.6 Arrays Could be Values . . . 23
2.4 Linked List . . . 24
2.4.1 Design Issues . . . 24
2.4.2 Preexisting Linked-List Libraries . . . 24
2.4.3 Link Attachment: Intrusive *vs*. Wrapped . . . 24
2.4.4 Axis: Single *vs*. Multi-Static *vs*. Dynamic . . . 26
2.4.5 User Integration: Preprocessed *vs*. Type-System Mediated . . . 30
2.4.6 List Identity: Headed *vs*. Ad-Hoc . . . 31
2.4.7 End Treatment: Uniform *vs*. Cased . . . 31
2.5 String . . . 33
2.5.1 Character types . . . 35

3 Array **39**
3.1 Introduction . . . 39
3.2 Features Added . . . 41
3.3 Single Dimension Array Implementation . . . 45
3.4 Typing of C Arrays . . . 48
3.5 Multidimensional Array Implementation . . . 53
3.6 Zero Overhead . . . 58
3.7 Array Lifecycle . . . 64
3.8 Array Comparison . . . 67
3.8.1 Rust . . . 67
3.8.2 Java . . . 69
3.8.3 C++ . . . 72
3.8.4 Subscript *vs*. iteration . . . 73
3.9 Future Work . . . 74
3.9.1 Array Syntax . . . 74
3.9.2 Error Messages . . . 75
3.9.3 Retire Pointer Arithmetic . . . 75
3.9.4 Defer Size Tracking . . . 76
3.9.5 Performance Benchmarking and Case Studies . . . 77
3.9.6 Guarantee Zero Overhead . . . 77

4 Linked List **79**
4.1 Plan-9 Inheritance . . . 79
4.2 Features . . . 80
4.2.1 Core Design Issues . . . 80
4.3 List API . . . 84
4.3.1 Iteration . . . 86
4.4 C++ Lists . . . 87
4.5 Implementation . . . 88
4.6 Assessment . . . 89
4.6.1 Experiment Design . . . 90
4.6.1.1 Add-Remove Performance . . . 90
4.6.1.2 Test Program . . . 92
4.6.1.3 Use Cases . . . 93
4.6.1.4 Sizing . . . 94
4.6.1.5 Execution Environment . . . 96
4.6.1.6 Recap and Master Legend . . . 96
4.6.2 Result: Coarse Comparison of Styles . . . 97
4.6.3 Result: Intrusive Winners and Losers . . . 99
4.6.4 Result: Sweet and Sore Spots . . . 101
4.7 Future Work . . . 102
4.7.1 Bounded Polymorphism . . . 102
4.7.2 Performance Assessment . . . 102

5 String **105**
5.1 String Operations . . . 105
5.2 C∀ string Type . . . 106
5.2.1 Implicit String Conversions . . . 107
5.2.2 Length . . . 108
5.2.3 Comparison Operators . . . 108
5.2.4 Concatenation . . . 108
5.2.5 Repetition . . . 109
5.2.6 Substring . . . 109
5.2.7 Searching . . . 110

5.2.8 Returning N on Search Failure . . . 112
5.2.9 C Compatibility . . . 113
5.2.10 I/O Operators . . . 113
5.2.11 Assignment . . . 115
5.2.12 Logical Overlap . . . 118
5.3 Storage Management . . . 121
5.3.1 Individual String . . . 121
5.3.2 String Heap . . . 123
5.3.3 RAII Limitations . . . 125
5.3.4 Sharing Implementation . . . 128
5.3.5 Controlling Implicit Sharing . . . 129
5.3.6 Sharing and Threading . . . 129
5.3.7 Short-String Optimization . . . 131
5.4 Performance Assessment . . . 131
5.4.1 Methodology . . . 131
5.4.2 Test: Append . . . 132
5.4.2.1 Fresh vs Reuse in C++, Emulation Baseline . . . 132
5.4.2.2 C∀'s Sharing Mode . . . 133
5.4.2.3 Misusing Concatenation . . . 133
5.4.3 Test: Pass Argument . . . 136
5.4.4 Test: Allocate . . . 136
5.5 Unicode character support . . . 141
5.6 Future Work . . . 144

6 Conclusion **145**
6.1 Arrays . . . 145
6.2 Lists . . . 145
6.3 Strings . . . 146
6.4 Future Work . . . 146

References **147**

List of Figures

2.1 Pre-VLA contiguous matrix styles, fixed *vs.* variable . . . 18
2.2 C99 matrix styles, contiguous *vs.* non-contiguous . . . 19
2.3 Multiple ways to declare an array parameter . . . 21
2.4 Three styles of link attachment . . . 25
2.5 Simulation of wrapped using intrusive . . . 27
2.6 Example of simultaneity using LQ lists . . . 28
2.7 Comparison of headed and ad-hoc list identities . . . 32
2.8 LQ field-level representation of links and ends . . . 33
2.9 Element-relative insertion operations on LQ's uniform-first cased-last representation. . . . 34

3.1 Example of calling a dimension-generic function . . . 42
3.2 N-style parameters, for C++ template *vs.* C∀ generic type . . . 43
3.3 C∀ Dynamic Array Member (DAM) declaration . . . 44
3.4 C∀ Dynamic Array Member (DAM) usage . . . 45
3.5 Case comparison for array type compatibility . . . 52
3.6 Visualization of subscripting by value and by 'all' . . . 56
3.7 Definition of structure 'arpk,' which underlies the C∀ array . . . 58
3.8 Bound-check overhead comparison, control case . . . 61
3.9 Overhead comparison, differentiation case, source. . . . 62
3.10 Overhead comparison, differentiation case, assembly. . . . 63
3.11 Exponential thunk generation under the otype-recursion pattern . . . 65
3.12 Triangular matrix in Java and C . . . 70
3.13 Mostly iterator-based matrix multiplication in Rust . . . 74

4.1 Plan-9 polymorphism . . . 80

4.2 Diamond non-virtual inheritance pattern . . . 81
4.3 Multiple link axes in C∀ list library . . . 81
4.4 Sketch of the dlink and dlist type definitions . . . 82
4.5 Demonstration of multiple static link axes in C∀ . . . 83
4.6 Iterator driver . . . 86
4.7 Linked list iteration examples . . . 87
4.8 Obtaining a linked-list iterator in C++ *vs.* C∀ . . . 88
4.9 C∀ list library representations for headed and headless lists . . . 89
4.10 Experiment use cases, numbered . . . 94
4.11 Variety of IR duration responses to list length, at small–medium lengths . . . 95
4.12 IR duration, transformed for general analysis . . . 97
4.13 IR duration *vs.* list length, all sizes . . . 98
4.14 IR duration, decomposed by all first-order effects . . . 100
4.15 IR duration where framework selection interacts with other factors . . . 101

5.1 Inter-language comparison of string APIs . . . 106
5.2 Simplification by C∀ API for extracting words from a line of text . . . 112
5.3 Passing substrings as parameters, program . . . 121
5.4 Passing substrings as parameters, execution . . . 122
5.5 String memory-management data structures . . . 123
5.6 Code lowering by RAII implementation . . . 127
5.7 Controlling copying *vs.* sharing of strings . . . 130
5.8 Fresh *vs.* reuse in C++, emulation baseline . . . 134
5.9 C∀ compromise for fresh *vs.* reuse . . . 134
5.10 C∀'s low overhead for misusing concatenation . . . 135
5.11 Impact of C++ small-string optimization on experimental timings . . . 137
5.12 Memory-allocation test harness and its usage pattern . . . 138
5.13 String allocation space and time performance . . . 139
5.14 CFA program illustrating UTF-8 support . . . 141

List of Tables

2.1 Syntactic reference for array *vs.* pointer 10
2.2 Syntactic reference for decay during parameter passing 15

4.1 Glossary of terms used in the list performance evaluation 91

5.1 Comparison of languages' strings, storage management perspective 116
5.2 HL to LL C∀ string-API lowering 128

Chapter 1

Introduction

Practically all modern programming languages provide at least these three high-level collections (containers): array, linked-list, and string. In the imperative tradition, the array is part of the programming language, while linked lists are built from (recursive) pointer types, often via standard-library help. The functional tradition leans heavily on linked lists, which are often helped by a language-provided notation, while, typically, a library offers an array capability via an API. Occasionally, a higher-level language's focus leads to omitting direct support for a pedantic definition of array. But there, the language's interoperability encourages implementing high-performance kernels elsewhere, while the language's own pseudo-array facility (*e.g.*, wide tree) still captures much of the expected difference of an array, compared with a linked list. Strings are generally built from arrays and/or linked lists; any given language's documentation will addresses the question, "How do *we* treat text?" For all three types, languages and/or their libraries supply varying degrees of high-level mechanisms for manipulating these objects at the bulk and component levels, such as copying, slicing, extracting, iterating among, adding and removing elements.

Unfortunately, memory misuse under C-idiomatic programming causes a significant number of errors [55]. Estimates suggest 50% [37] of total reported open-source vulnerabilities occurring in C result from errors using these facilities (memory errors). For operating system and browser vendors, who heavily use systems languages, 60%–70% of reported software vulnerabilities involved memory errors [31]. For Microsoft, 70% of vulnerabilities addressed via security updates between 2006–2018 are memory safety issues [38, slide 10]. In a study of software exploits in the U.S. National Vulnerability Database over 2013–2017, the top reported vulnerability is buffer errors (in the sense of misused array bounds), among 19 vulnerability categories [12].

While these figures address memory errors, not limited to the three primary collections, the authors' explanations of memory errors suggest improving on C's language/library options for working with the three collections would be beneficial. Every one of the safety-statistic sources above includes array bounds in its first characterization of unsafe programming-language features. Those studies that deal exclusively with memory safety (*i.e.*, the four excluding [12]) all put malloc–free upfront as well. The specific survey of C features, [55], places further blame on:

- the null-terminated text idiom, with its support/mandate in the C standard library and
- the unprotected nature of C pointers.

An effort at translating C code to the safe subset of the now-famously safe language Rust [15] shows pointers as the stand-out difficult aspect to analyze. It demonstrates the difficulty of auto-

matically determining pointer-referent lifetimes in that its success claim is to reduce the occurrence rate of unsafe dereference operations to about once per 100 lines of code.

A tool for analyzing C code's use of linked data structures (including, *e.g.*, skip list and binary tree) found variations of the linked-list shape to be present in most of the programs in its corpus [57].

So, C's array unsafety is infamous, its string pattern necessitates explicit storage management (also infamous), and user-written linked lists are a recognizable player in the arena of (infamously) unsafe raw pointer uses. Therefore, hardening the three primary collections goes a long way to making the majority of C programs safer, eliminating major hacker attack vectors.

My work looks at extending these foundational collection types in the programming language C∀, which is a new dialect of the C programming language. A primary goal of C∀ [11] is 99% backward compatibility with C, while maintaining a look and feel that matches with C programmer experience and intuition. This goal requires "thinking inside the box" to engineer new features that "work and play" with C and its massive legacy code-base. An equally important goal is balancing good performance with safety.

The thesis describes improvements I made to the C∀ language design, both syntax and semantics, to support the collection features, and the source code created within the C∀ compiler, libraries, and runtime system to implement these features. This work leverages preexisting work within the compiler's type and runtime systems generated by prior graduate students working on the C∀ project.

1.1 Array

An array provides a homogeneous collection with $O(1)$ access to elements using subscripting. Higher-level operations like array slicing (single or multidimensional) may have significantly higher cost, but provide a better programmer experience. The array size can be static, dynamic but fixed after creation, or dynamic and variable after creation. For static and dynamic-fixed, an array can be stack allocated, while dynamic-variable requires the heap. Because array layout has contiguous components, subscripting is a computation, *i.e.*, some form of pointer arithmetic. Not included in this requirement are pseudo-arrays with arbitrary keys, *e.g.*, hash table subscripted by a string key (see below).

C provides a simple array type as a language feature. However, it adopts the controversial position of treating pointers and arrays as twins, leading to multiple problems. Hence, the way that arrays are typically passed around in a program removes the information necessary to do bound checks.

1.2 Linked List

A linked-list provides a homogeneous collection often with $O(n)$ access to elements using successor and predecessor operations that involve pointer chasing. It is the most foundational case of the linked data structures, which include trees and hash tables. Among the linked structures, the

list's defining characteristic is maintaining a user-explicit total order (chain shape). Element search can be by position (as for array) or by value; it is sometimes presented as a subscript. In a more complex structure, the linked shape may be system managed (user implicit) by being a function of a designated "key" datum, *e.g.*, often for a hash table. Though such schemes often give better-than-$O(n)$ lookup, the linked list's user-explicit shape limits what the system can provide, to fast step/iteration for user-"nearby" data, and slow exhaustive search/seek otherwise. Linked types are dynamically sized by adding and removing nodes using link fields internal or external to the list elements (nodes). If a programming language allows pointers to stack storage, linked nodes can be allocated on the stack and connected with stack addresses; otherwise, elements are heap allocated with internal or external link fields.

A typical C distribution provides a few simple, library data-structures:

- singly-linked lists, singly-linked tail queues, lists, and tail queues (<sys/queue.h>) (see Section 2.4.2, p. 24)
- hash search table consisting of a key (string) with associated data (<search.h>)

Because these collection libraries can be restrictive or awkward to use, and they are not universally portable, C programmers often build bespoke linked data-structures, which further increases the potential for memory errors.

1.3 String

A string provides a dynamic array of homogeneous elements, where the elements are (often) some form of human-readable characters. What differentiates a string from other types is that many of its operations work on groups of elements for scanning and changing, *e.g.*, index and substr. While subscripting individual elements is usually available, working at the character level is considered poor practice, *i.e.*, underutilizing the powerful string operations. Therefore, the cost of a string operation is usually less important than the power of the operation to accomplish complex text manipulation, *e.g.*, search, analyzing, composing, and decomposing string text. A string can take on an arbitrary size unknown until runtime, so storage is normally heap allocated but often implicitly managed, even in unmanaged languages. In many cases, string management is separate from heap management, *i.e.*, string libraries roll their own heaps.

The C string type is just a character array and requires user storage management to handle varying size. C adopts a terminating sentinel character, `'\0'`, to mark the end of a variable-length character array, as opposed to keeping an independent length field. Hence, the sentinel character is excluded as an element within a string and determining the string length is an $O(N)$ operation. The C standard library includes a number of high-level operations for working with this representation. Most of these operations are awkward to use and highly error prone.

1.4 Motivation

The primary motivation for this work is twofold:

1. The three primary collection types in C are difficult to understand, teach, and get right because they are too low level. Providing higher-level, feature-rich versions of these collections in C∀ is a major component of the primary goal. The result is a simplified programming experience, which increases productivity and maintainability.
2. The new collection types must be as correct and safe to use as those in other modern programming languages, to address the concerns of industry, government, and military [16]. Prior approaches focus on out-of-bound array accesses using a model-based approach (ASCET) in embedded systems (*e.g.*, cars) [6], and general and null-terminated string arrays using a C++ template syntax (Checked C) [14, 46]. Both White House [18] and DARPA [3] recently released a recommendation to *move away* from C and C++, because of cybersecurity threats exploiting vulnerabilities in these languages. Fixing these vulnerabilities negates this need, allowing C and its ecosystem to continue into the future.

While new languages, *e.g.*, Go, Rust, Swift, purport to be systems languages replacing C, the reality is that rewriting massive C code-bases is impractical and often a non-starter if the runtime uses garbage collection [7]. Even assuming automated conversion of C programs to safe languages, who will maintain this massive new code base? Furthermore, new languages must still interact with the underlying C operating system through fragile, type-unsafe, interlanguage-communication. Switching to C++ is equally impractical as its complex and interdependent type-system (*e.g.*, objects, overloading, inheritance, templates) means idiomatic C++ code is difficult to use from C, and C programmers must expend significant effort learning new C++ features and idioms. Finally, projects claiming to be modern C replacements (Cyclone [20], Polymorphic C [49], GObject [17]) do not retain C backwards compatibility in syntax, programming model, or semantic compatibility; these languages are equivalent to switching to a different programming language. Hence, rewriting and retraining costs for other languages are prohibitive when companies have a large C software-base (Google, Apple, Microsoft, Amazon, AMD, Nvidia).

1.5 Which C?

Like many established programming languages, C has a standards committee and multiple ANSI/-ISO language manuals [24, 25, 26, 27]. However, most programming languages are only partially explained by their standard's manual(s). When it comes to explaining how C works, an unavoidable source of truth is the gcc compiler, which is mimicked by other C compilers, such as Clang [13], because a large part of the C library (runtime) system (glibc on Linux) contains gcc features. Some key aspects of C need to be explained and understood by quoting from the language reference manual. However, to illustrate actual program semantics, this thesis constructs multiple small programs whose behaviour exercises a particular point and then confirms the behaviour by running the program using multiple gcc compilers. These example programs show

- whether the compiler accepts or rejects certain syntax,
- printed output to buttress a behavioural claim, or
- passed/failed assertions testing pre/post-conditions or invariants.

These programs are tested across gcc versions 8–14 and clang versions 10–14 running on ARM, AMD, and Intel architectures. Any discovered anomalies among compilers, versions, or architectures are discussed.

In general, it is never clear whether the *truth* lies in the C standard or the compiler(s), which may be true for other programming languages. This work is not out to argue for the compiler(s) being truer than the standard. But it often assumes the pragmatic position that a compiler demonstration is more effective than document interpretation at illustrating a telling case. Having established a leading compiler's handling, which is central to C programmer experience, this thesis usually moves on, without attempting to discern whether the standard judges this handling as intended, or as an implementation quirk.

1.6 Contributions

Overall, this work has produced significant syntactic and semantic improvements to C's arrays, linked-lists and string types. The following are my detailed contributions, where performance and safety are always the motivating factors.

1.6.1 Array

The improvements to C arrays are:

1. Introduce a small number of subtle changes to the typing rules for the C array, while still achieving significant backwards compatibility.
2. Create a new polymorphic mechanism in the C∀ **forall** clause to specify array dimension values, similar to a C++ non-typed template parameter.
3. Construct a new standard-library array type, available through **#include** <array.hfa>.

The new array type accesses prior C∀ type-system features, delivering guaranteed runtime bound checks (often optimizer-removable) and implicit (guaranteed accurate) inter-function length communication. Extending the preexisting C∀ type system to achieve this length-related type checking is an original contribution. Furthermore, the new array incorporates multidimensional capabilities typical of scientific/machine-learning languages, made to coexist with the C raw-memory-aware tradition, in a novel way. The thesis includes a performance-relevant assessment of generated code that shows the new C∀ array simultaneously delivering the performance of a C array and the safety of a C++ vector.

1.6.2 Linked List

The linked list is a new runtime library, available through **#include** <list.hfa>. The library offers a novel combination of the preexisting C∀ features: references, inline inheritance, polymorphism, and declaration scoping. A programmer's experience using the list data types within the type system is novel. The list's runtime representation follows the established design pattern of intrusive link fields for performance reasons, especially in concurrent programs. The thesis includes a performance evaluation that shows the list performing comparably with other C-family intrusive lists, and notably better than the C++ standard list.

1.6.3 String

A new string type and its runtime library are available through **#include** <string.hfa>. The library offers a type where basic usage is comparable to the C++ string type, including analogous coexistence with raw-character pointers. Its implementation, however, follows different principles, enabling programs to work with strings as simple values, without incurring excessive copying. Mutability is retained. Substrings are supported, including the option for overlapping ranges to share edits transparently, or act as diverging snapshots. The implementation allows a set of strings to use an optimized, shared, custom heap, or to use the regular heap, on strings that are passed among threads.

The string library includes reading and writing strings via the preexisting C∀ I/O stream library. Enabling transparent reading of unknown length strings using managed allocation required revision of the stream library's existing handling of character arrays. All reasonable stream manipulators are supported, *e.g.*, width, justification (left/right), printing characters as their numeric values, and complex input scanning to isolate strings in the input stream.

Chapter 2

Background

Since this work builds on C, it is necessary to explain the C mechanisms and their shortcomings for array, linked list, and string.

Much of this discussion is critical of C. So, a concern about judging historic decisions by modern standards is understandable. But this discussion did not create the situation with which the world is currently dealing: strong ties to an arguably historic codebase that must withstand modern threats, being pervasive in the modern tech stack. C∀ is a response to this situation. This discussion aims to be precise about how C, as-is, falls short of the need, as-is. Thereby, the present contributions, which improve C∀, are properly contextualized.

Indeed, a particular shortcoming discussed herein may be inevitable, in absence of features that pre-existing C∀ added, beyond C's. For example, the C linked list is criticized for its dependence on textual macros; but C does not offer polymorphism, which is practically necessary for doing better. So, the criticism of this criterion is not claiming that the original C linked-list designers ought to have made a non-macro decision, but rather, that the drawbacks of a macro-based solution, which are now avoidable, are therefore worth identifying.

2.1 Ill-Typed Expressions

C traditionally reported many ill-typed expressions as warnings. For example, these attempts to assign y to x and vice-versa are ill-typed.

```
float * x;                          // x points at a floating-point number
void (*y)(void);                    // y points at a function
x = y;                              // wrong
y = x;                              // wrong
```

with warnings:

```
warning: assignment to 'float *' from incompatible pointer type 'void (*)(void)'
warning: assignment to 'void (*)(void)' from incompatible pointer type 'float *'
```

Similarly,

```
float pi = 3.14;
void f( void (*g)(void) );
f( &pi );                           // wrong
```

with warning:

```
warning: passing argument 1 of 'f' from incompatible pointer type
note: expected 'void (*)(void)' but argument is of type 'float *'
```

Clearly, GCC understands these ill-typed cases, and yet allows the program to compile, which seems inappropriate.

GCC 13+ reports this condition as an error by default, rather than a warning. GCC always supported -W(no-)error=X, which makes it report the selected condition X as an error (*vs.* warning, for no-error); so, it was up to the user to decide how severe a condition is. Previously this specific condition was a warning, now it is an error. C∀'s type system also rejects programs with this condition, reporting a type-mismatch error.

A program can be ill-typed, regardless of the (default) reporting being error *vs.* warning.

Still, not all warnings are created equally. Sometimes, GCC reports a warning on one case, but the warning detection fails to trigger on a related case where the problem is still present. GCC's behaviour for such an issue is one of best effort. Array length is an issue particularly relevant to this thesis; GCC is continually shipping with better array-length issue detectors. However, even at gcc-16 -Wall -Wextra -O3, the detection is fallible to simple obfuscations, such as rewriting a length from a literal to a constant.

```
void f( float[7] );
static const int n = 5;
void g( float x[5], float y[n] ) {
    f( x );  // warning
    f( y );  // no warning
}
```

The issue here is soundness, whether the absence of an issue report is sufficient grounds to conclude that the issue type is absent.

In the following discussion, an *ill-typed* program fragment is one that gets an error or warning, with a message that discusses typing, where the detection pattern gives the user has a reasonable expectation of soundness. After the GCC-13 -Werror fix, there is a strong correlation between best-effort detection and "just a warning." On the topic of array length, this thesis is mainly interested in protecting uses with sound rules; it will say that GCC ignores a certain issue (or, accepts a program with it), to mean that the issue has no sound check.

2.2 Reading Declarations

A significant area of confusion is reading C declarations, which results from interesting design choices.

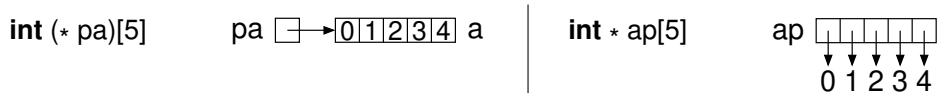
- In C, it is possible to have a value and a pointer to it.

  ```
  int i = 3, * pi = &i;
  ```

  Extending this idea, it should be possible to have an array of values and pointer to it.

  ```
  int a[5] = { 1, 2, 3, 4, 5 },  * pa[5] = &a;
  ```

However, the declaration of pa is incorrect because dimension has higher priority than pointer, so the declaration means an array of 5 pointers to integers. The declarations for the two interpretations of * [5] are:



If the priorities of dimension and pointer were reversed, the declarations become more intuitive: **int** * pa[5] and **int** * (ap[5]).

- This priority inversion extends into an expression between dereference and subscript, so usage syntax mimics declaration.

```
int (* pa)[5]          int * ap[5]
    (*pa)[i] += 1;         *ap[i] += 1;
```

(Section 2.3.1, p. 12 shows pointer decay allows the first form to be written pa[i] += 1, which is further syntax confusion.) Again, if the priorities were reversed, the expressions become more intuitive: *pa[i] += 1 and *(ap[i]) += 1. A similar priority inversion exists between deference '*' and field selection '.' (period), so *ps.f means *(ps.f); this anomaly is *fixed* with operator ->, which performs the two operations in the more intuitive order: sp->f ⇒ (*sp).f.

While attempting to make the declaration and expression contexts consistent is a laudable goal, it has not worked out in practice, Dennis Richie's contrary position on the issue conceded:

> In spite of its difficulties, I believe that the C's approach to declarations remains plausible, and am comfortable with it; it is a useful unifying principle. [44, p. 12]

As it stands, typed must be read from their declared identifier outward, following the "clockwise spiral rule" [5]. Unfortunately, C∀ cannot correct these operator priority inversions without breaking C compatibility.

The alternative solution is for C∀ to provide its own, more intuitive declaration syntax for types, variables and routines. The new declarations place type operators to the left of the base type, while C declarations place them (to the right of the base type and) spiraling around the declared object name. The type operators have the same semantics in C∀ as in C, so there is nothing to learn. Then, a C∀ declaration is read left to right, where a function return-type is enclosed in brackets [].

C	C∀	read left to right
int * x1 [5];	[5] * **int** x1;	*// array of 5 pointers to int*
int (*x2)[5];	* [5] **int** x2;	*// pointer to array of 5 int*
int (*f(**int** p))[5];	[* [5] **int**] f(**int** p);	*// function returning pointer to array of 5 ints*
		// and taking an int argument

As declaration size increases, it becomes correspondingly difficult to read and understand the C form because each time a spiral is encountered, the reader's mental flow is broken. Note that writing declarations left to right is common in other programming languages, where the function return-type is often placed after the parameter declarations, *e.g.*, C++ **auto** f(**int**) -> **int**. However, putting the return type at the end deviates from where the return value logically appears in an expression, x = f(...) versus f(...) = x. Interestingly, programmers normally speak a declaration from left to right, regardless of how it is written. While this "left to right" explanation is Latin-centric

Table 2.1: Syntactic reference for array *vs*. pointer

	Description	C	C∀	C∀-thesis
▷	value	T x;	T x;	
	immutable value	**const** T x; T **const** x;	**const** T x; T **const** x;	
▷	pointer to value	T * x;	* T x;	
	immutable ptr. to val.	T * **const** x;	**const** * T x;	
	ptr. to immutable val.	**const** T * x; T **const** * x;	* **const** T x; * T **const** x;	
▷	array of value	T x[10];	[10] T x	array(T, 10) x
	ar. of immutable val.	**const** T x[10]; T **const** x[10];	[10] **const** T x [10] T **const** x	**const** array(T, 10) x array(T, 10) **const** x
	ar. of ptr. to value	T * x[10];	[10] * T x	array(T *, 10) x array(* T, 10) x
	ar. of imm. ptr. to val.	T * **const** x[10];	[10] **const** * T x	array(* **const** T, 10) x array(**const** * T, 10) x
	ar. of ptr. to imm. val.	**const** T * x[10]; T **const** * x[10];	[10] * **const** T x [10] * T **const** x	array(**const** T *, 10) x array(* **const** T, 10) x
▷	ptr. to ar. of value	T (*x)[10];	* [10] T x	* array(T, 10) x
	imm. ptr. to ar. of val.	T (* **const** x)[10];	**const** * [10] T x	**const** * array(T, 10) x
	ptr. to ar. of imm. val.	**const** T (*x)[10]; T **const** (*x)[10];	* [10] **const** T x * [10] T **const** x	* **const** array(T, 10) x * array(T, 10) **const** x
	ptr. to ar. of ptr. to val.	T *(*x)[10];	* [10] * T x	* array(T *, 10) x * array(* T, 10) x

(like C), the more inclusive generalization, "in time order," applies: the reader's time-ordered experience of the written form is literally meaningful in the C∀ notation, but violated in the C notation.

Specifically, Table 2.1 introduces the many layers of the C and C∀ array story, where the C∀ story is discussed in Chapter 3, p. 39. The C∀-thesis column shows the new array declaration form, which is my contribution to safety and ergonomics. The table shows there are multiple yet equivalent forms for the array types under discussion, and subsequent discussion shows interactions with orthogonal (but easily confused) language features. Each row of the table shows alternate syntactic forms. The simplest occurrences of types distinguished in the preceding discussion are marked with ▷. Removing the declared variable x, gives the type used for variable, structure field, cast, or error messages. Unfortunately, parameter declarations have more syntactic forms and rules (see Section 2.3.1 page 13).

2.3 Array

A key design decision with respect to C's arrays is also a key contributor to C's unsafety as experienced today. As Kerninghan and Ritchie stated in 1978:

> In C, there is a strong relationship between pointers and arrays, strong enough that pointers and arrays really should be treated simultaneously. Any operation which can be achieved by array subscripting can also be done with pointers. [32, p. 93]

Accessing any storage requires pointer arithmetic, even if it is just base-displacement addressing in an instruction. The conjoining of pointers and arrays could also be applied to structures, where a pointer references a structure field like an array element. Finally, while subscripting involves pointer arithmetic (as does a field reference x.y.z), the computation is complex for multi-dimensional arrays and requires array descriptors to know stride lengths along dimensions. Many C errors result from manually performing pointer arithmetic instead of using language subscripting so the compiler performs the arithmetic.

Unfortunately, C semantics want a programmer to *believe* an array variable is a *pointer to its first element*. This desire becomes apparent by a detailed inspection of an array declaration.

```
float ar[10];
```

The inspection begins by using **sizeof** to provide program semantics for the intuition of an expression's type. An architecture with 64-bit pointer size is used, to remove irrelevant details.

```
static_assert( sizeof(float) == 4 );      // floats (array elements) are 4 bytes
static_assert( sizeof(void *) == 8 );     // pointers are 8 bytes
```

Now consider the **sizeof** expressions derived from ar, modified by adding pointer-to and first-element (and including unnecessary parentheses to avoid any confusion about precedence).

```
static_assert( sizeof(ar) == 40 );        // array
static_assert( sizeof(&ar) == 8 );        // pointer to array
static_assert( sizeof(ar[0]) == 4 );      // first element
static_assert( sizeof(&(ar[0])) == 8 );   // pointer to first element
```

Given that arrays are contiguous and the size of **float** is 4, then the size of ar with 10 floats being 40 bytes is common reasoning for C programmers. Equally, C programmers know the size of a pointer to the first array element is 8. Clearly, an array and a pointer to its first element are different.

In fact, the idea that there is such a thing as a pointer to an array may be surprising. It it is not the same thing as a pointer to the first element.

```
typeof(&ar) x = &ar;                      // x is pointer to array
typeof(&(ar[0])) y = &ar[0];              // y is pointer to first element
x = y;                                    // ill-typed
y = x;                                    // ill-typed
```

The first assignment generates:

```
warning: assignment to `float (*)[10]' from incompatible pointer type `float *'
```

and the second assignment generates the opposite.

The inspection now refutes any suggestion that **sizeof** is informing about allocation rather than type information. Note that **sizeof** has two forms, one operating on an expression and the other on a type. Using the type form yields the same results as the prior expression form.

```
static_assert( sizeof(typeof(ar)) == 40 ); // array
static_assert( sizeof(typeof(&ar)) == 8 );   // pointer to array
static_assert( sizeof(typeof(ar[0])) == 4 ); // first element
static_assert( sizeof(typeof(&(ar[0]))) == 8 ); // pointer to first element
```

The results are also the same when there is no allocation at all, *e.g.*, a parameter declaration. This time, starting from a pointer-to-array type:

```
void f( float (*pa)[10] ) {
    static_assert( sizeof( *pa ) == 40 );  // array
    static_assert( sizeof( pa ) == 8 );    // pointer to array
    static_assert( sizeof( (*pa)[0] ) == 4 ); // first element
    static_assert( sizeof(&((*pa)[0])) == 8 ); // pointer to first element
}
f( &ar );
```

Hence, in all cases, **sizeof** is reporting on type information.

Therefore, thinking of an array as a pointer to its first element is too simplistic an analogue and it is not backed up by the type system. This misguided analogue works for single-dimension arrays but there is no advantage other than possibly teaching beginner programmers about basic runtime array-access.

Continuing, there is a short form for declaring array variables using length information provided implicitly by an initializer.

```
float fs[] = {3.14, 1.77};
char cs[] = "hello";                    // shorthand for { 'h', 'e', 'l', 'l', 'o', '\0' }
static_assert( sizeof(fs) == 2 * sizeof(float) );
static_assert( sizeof(cs) == 6 * sizeof(char) );  // 5 letters + 1 null terminator
```

The compiler counts the number of initializer elements and uses this value as the first dimension. Unfortunately, the implicit element counting does not extend to dimensions beyond the first.

```
float fm[][2] = { {3.14, 1.77}, {12.4, 0.01}, {7.8, 1.23} };  // brackets define structuring
char cm[][sizeof("hello")] = { "hello", "hello", "hello" };
static_assert( sizeof(fm) == 3 * 2 * sizeof(float) );
static_assert( sizeof(cm) == 3 * 6 * sizeof(char) );
```

My observation is recognizing:

- There is value in using a type that knows its size.
- The type pointer to the (first) element does not.
- C *has* a type that knows the whole picture: array, *e.g.*, T[10].
- This type has all the usual derived forms, which also know the whole picture. A noteworthy example is pointer to array, *e.g.*, T (*)[10].

2.3.1 Arrays Decay and Pointers Diffract

The last section established the difference among these four types:

```
float ar[10];                   // array
float (*pa)[10] = &ar;          // pointer to array
float a0 = ar[0];               // element
float * pa0 = &(ar[0]);         // pointer to element
```

But the expression used for obtaining the pointer to the first element is pedantic. Much C programmer experience with arrays stems from the shortcut

```
float * pa0x = ar;                    // (ok)
```

which reproduces pa0, in type and value:

```
assert( pa0 == pa0x );
```

The validity of this initialization is unsettling, in the context of the facts established in Section 2.3, p. 10. Notably, it initializes name pa0x from expression ar, when they are not of the same type:

```
assert( sizeof(pa0x) != sizeof(ar) );
```

So, C provides an implicit conversion from **float**[10] to **float** *.

> Except when it is the operand of the **sizeof** operator, or the unary & operator, or is a string literal used to initialize an array, an expression that has type "array of *type*" is converted to an expression with type "pointer to *type*" that points to the initial element of the array object [25, § 6.3.2.1.3]

This phenomenon is the famous (infamous) *pointer decay*, which is a decay of an array-typed expression into a pointer-typed one. It is worthy to note that, in the expression ar[i], the subexpression ar is not one of the standard's exceptions; ar is converted to a pointer before subscripting occurs. Thus, *all* subscripting happens on pointers, never arrays.

After this pointer decay (when needed), subscripting proceeds. In the next step, [25, § 6.5.2.1.2] explains that ar[i] is treated as if it is (*(ar+i)). [25, § 6.5.6.8] explains that the addition, of a pointer with an integer type, is defined only when the pointer refers to an element that is in an array. The addition gives the address i elements away from the element (ar, or &ar[0]). This address is valid if ar is big enough and i is small enough. Finally, [25, § 6.5.3.2.4] explains that the * operator's result is the referenced element. Taken together, these rules illustrate that ar[i] and i[ar] mean the same thing, as integer addition is commutative.

Subscripting a pointer when the target is standard-inappropriate is still, practically, well-defined. While the standard affords a C compiler freedom about the meaning of an out-of-bound access, or of subscripting a pointer that does not refer to an array element at all, the fact that C is famously both generally high-performance, and specifically not bound-checked, leads to an expectation that the runtime handling is uniform across legal and illegal accesses. Moreover, consider the common pattern of subscripting on a malloc result:

```
float * fs = malloc( 10 * sizeof(float) );
fs[5] = 3.14;
```

The malloc behaviour is specified as returning a pointer to "space for an object whose size is [the size requested]" ([25, § 7.22.3.4.2]). But *nothing* more is said about this pointer, much less that its referent might *be* an array, *i.e.*, that subscripting might be allowed at all.

Under this assumption, a pointer, p, being subscripted (or added to, then dereferenced) by any value (positive, zero, or negative), gives a view of the program's entire address space, centred around p's address, divided into adjacent **sizeof**(*p) chunks, each potentially (re)interpreted as **typeof**(*p). I call this phenomenon *array diffraction*, which is a diffraction of a single-element pointer into the assumption that its target is conceptually in the middle of an array, whose size is unlimited in both directions, *e.g.*, (&ar[5])[−200] or (&ar[5])[200].

No pointer is exempt from array diffraction. No array shows its elements without pointer decay.

A further pointer–array confusion, closely related to decay, occurs in parameter declarations. [25, § 6.7.6.3.7] explains that when an array type is written for a parameter, the parameter's type changes into (what is practically) the pointer-decayed type. But pointer decay applies to an expression, while this transformation applies to a declaration. The respective handling of the following two ways of writing a parameter shows that the array and pointer versions are identical.

```
void f( float x[10], float * y ) {
    static_assert( sizeof(x) == sizeof(void *) );
    static_assert( sizeof(y) == sizeof(void *) );
    x = y;  y = x;
}
```

As the **sizeof**(x) meaning changed, compared with when run on a similarly written local variable declaration, GCC also gives this code the warning for the first assertion:

```
warning: 'sizeof' on array function parameter 'x' will return size of 'float *'
```

The caller of such a function is left with the reality that a pointer parameter is a pointer, no matter how it is written:

```
// reusing local variable float ar[10]
float v;
f( ar, ar );                    // ok: two decays, one into an array spelling
f( &v, &v );                    // ok: no decays; a non-array passes to an array spelling
```

This fragment gives the warning for the first argument, in the second call.

```
warning: 'f' accessing 40 bytes in a region of size 4
```

The shortened parameter syntax T x[] is a further way to write *pointer*. Note the opposite meaning of this form now, compared with its use in local variable declarations. This point of confusion is illustrated in:

```
char ca[] = "hello";            // array on stack, initialized from read-only data
char * cp = "hello";            // pointer to read-only data [decay here]
void edit( char c[] ) {         // parameter is pointer
    c[3] = 'p';
}
edit( ca );                     // ok [decay here]
edit( cp );                     // Segmentation fault
edit( "hello" );                // Segmentation fault [decay here]
```

Note that C++ gives a warning for the initialization of cp.

```
warning: ISO C++ forbids converting a string constant to 'char*'
```

and C gives a warning at the call of edit, if **const** is added to the declaration of cp.

```
warning: passing argument 1 of 'edit' discards 'const' qualifier from pointer target type
```

The basic two meanings, with a syntactic difference helping to distinguish, are illustrated in the declarations of ca *vs.* cp, whose subsequent edit calls behave differently. The syntax-caused confusion is in the comparison of the first and last lines, both of which use a literal to initialize an object declared by writing T x[ ]. But these initialized declarations get opposite meanings, depending on whether the object is a local variable or a parameter!

In summary, when a function is written with an array-typed parameter,

Table 2.2: Syntactic reference for decay during parameter passing. Includes interaction with **const**ness, where *immutable* refers to a restriction on the callee's ability.

	Description	Type	Parameter Declaration	C∀
▷	pointer to value	T *	T * x, T x[10], T x[],	* T x, [10] T x, [] T x,
	immutable ptr. to val.	T * **const**	T * **const** x, T x[**const** 10], T x[**const**],	**const** * T x [**const** 10] T x, [**const**] T x,
	ptr. to immutable val.	**const** T * T **const** *	**const** T * x, T **const** * x, **const** T x[10], T **const** x[10], **const** T x[], T **const** x[],	* **const** T x, * T **const** x, [10] **const** T x, [10] T **const** x, [] **const** T x, [] T **const** x,
▷	ptr. to ar. of val.	T(*)[10]	T (*x)[10], T x[3][10], T x[][10],	* [10] T x, [3][10] T x, [][10] T x,
	ptr. to ptr. to val.	T **	T ** x, T * x[10], T * x[],	** T x, [10] * T x, [] * T x,
	ptr. to ptr. to imm. val.	**const char** **	**const char** * argv[], *others elided*	[] * **const char** argv, *others elided*

- an appearance of passing an array by value is always an incorrect understanding,
- a dimension value, if any is present, is ignored,
- pointer decay is forced at the call site and the callee sees the parameter having the decayed type.

Pointer decay does not affect pointer-to-array types, because these are already pointers, not arrays. As a result, a function with a pointer-to-array parameter sees the parameter exactly as the caller does:

```
void decay( float x[10] ) {
    static_assert(sizeof(x) == sizeof(void *) );
}
static_assert( sizeof(ar) == 10 * sizeof(float) );
decay( ar );
```

```
void no_decay( float (*px)[10] ) {
    static_assert(sizeof(*px) == 10 * sizeof(float));
}
static_assert( sizeof(*pa) == 10 * sizeof(float) );
no_decay( pa );
```

Table 2.2 gives the reference for the decay phenomenon seen in parameter declarations.

2.3.2 Variable-Length Arrays

As of C99, the C standard supports a *variable length array* (VLA) [24, § 6.7.5.2.5], providing a dynamic-fixed array feature (see Section 3.1, p. 39). The C++ standard does not support VLAs, but g++ and clang provide a limited form of them. A VLA is used when the desired number of array elements is *unknown* at compile time.

```
size_t n;
scanf( "%d", &n );
double ar[n];
```

The array dimension is read from outside the program and used to create an array of size n on the stack. The VLA is implemented by the alloca routine, which bumps the stack pointer.

When stack allocation works, it gives a more efficient allocation and identical element-handling code. But stack-allocating an array is sometimes a bad idea, particularly when the array's size is unbounded and the stack size is small. A large VLA can overflow a stack that is not sized for it. Typically, stack sizes are limited, to guard against infinite recursion. For user-mode programs, stack size can be increased or made unlimited, just like the heap. This ability does not help with programs that have a great many stacks (*e.g.*, coroutines, user-level threads and pthreads, where stacks cannot grow, and where all of the maximally sized stacks must fit in the address space). Also, this ability does not apply inside the Linux kernel, where each user thread's kernel-mode stack is strictly limited, typically to 16 KB. In these situations, heap or static allocation is naturally preferable. So, a sweet spot for stack-allocating a VLA happens when the length is bounded but unknown statically, and when the allocation is happening frequently.

2.3.3 Multidimensional Arrays

It is common to treat multidimensional arrays as compositions of lower-diminsional arrays. For example, in linear algebra, the terminology of [34] calls them *rows* and *columns* for a matrix, adding *tubes* for the "not flat on page" dimension of a rank-3 tensor. (There is little terminology beyond three dimensions, and the third dimension's may seem arbitrary.) For a further domain, consider this acrostic poem.

```
ADMIT
UI
DRIFT:
IT
TYRANNIZES.
```

This grid of characters has *read text* going horizontally and the embedded *keywords*, "audit" and "dirty," running vertically.

Accordingly, one can decompose a higher-dimensional array into lower-dimensional components. The same linear-algebra taxonomy says a *fibre* is the 1-dimensional structure obtained by fixing all indices but one, and a *slice* is the 2-dimensional one that fixes all but two. In an English acrostic poem, one could pick out a row to obtain a fragment of poem text or pick a column for a

keyword. In classic Business Intelligence[1], picking a value for some factor(s) (*e.g.*, time = current year) and leaving others as "consider all values" is called *slicing* [33], regardless of how many factors are (un)specified.

In general, the dimensioning and subscripting for multidimensional arrays has two syntactic forms: m[r,c] or m[r][c].

Commonly, an array, matrix or cube is visualized (especially in mathematics) as a contiguous row, rectangle or block. This conceptualization is reinforced by subscript ordering, *e.g.*, m_{rc} for a matrix and c_{trc} for a cube. Few programming languages differ from the mathematical subscript ordering. However, computer memory is flat, and hence, array forms are structured in memory as appropriate for the runtime system. The closest representation to the conceptual visualization is for an array object to be contiguous, and the language structures this memory using pointer arithmetic to access the values using various subscripts. This approach still has degrees of layout freedom, such as row or column major order, *i.e.*, juxtaposed rows or columns in memory, even when the subscript order remains fixed. For example, programming languages like MATLAB, Fortran, Julia and R store matrices in column-major order since they are commonly used for processing column-vectors in tabular data sets but retain row-major subscripting to match with mathematical notation. In general, storage layout is hidden by subscripting, and only appears when passing arrays among different programming languages or accessing specific hardware.

Figure 2.1 shows two C90 approaches for manipulating a contiguous matrix. Note that C90 does not support VLAs. The fixed-dimension approach (left) uses the type system; however, it requires all dimensions except the first to be specified at compile time, *e.g.*, m[][6], allowing all subscripting stride calculations to be generated with constants. Hence, every matrix passed to fp1 must have exactly 6 columns but the row size can vary. The variable-dimension approach (right) ignores (violates) the type system, *i.e.*, the parameter type has no suggestion of multidimensionality and some acrobatics are required to obtain a well-typed call[2]. Subscripting is performed manually using pointer arithmetic in the macro sub.

Many languages allow multidimensional arrays-of-arrays, *e.g.*, Pascal and C++.

```
(* Pascal *)
var m : array[0..4, 0..4] of Integer;  (* matrix *)
type AT = array[0..4] of Integer;  (* array type *)
type MT = array[0..4] of AT;  (* array of array type *)
var aa : MT;  (* array of array variable *)
m[1][2] := 1;   aa[1][2] := 1 (* same subscripting *)
```

```
// C++
int m[5][5];

typedef vector< vector<int> > MT;
MT vv( 5, vector<int>( 5 ) );
m[1][2] = 1;  vv[1][2] = 1;
```

[1] A discipline of preparing digital data for reporting, in which a numeric quantity like dollars sold is stored and queried according to decompositions by many discretized orthogonal factors like business unit, customer industry and time.

[2] One may be tempted to phrase a call as fp2(4, 4, vm1), but this call is ill-typed. Argument vm1 could match parameter declarations **int** m[][4] or **int** (*m)[4]. But only the argument &vm1[0][0] relates vm1 to parameter type **int***. (As does its confusingly equivalent form, vm1[0].)

Fixed Dimension

```
void fp1( int rows, int m[][6] ) {
    ... printf( "%d ", m[r][c] ); ...
}
int fm1[4][6], fm2[6][6]; // no VLA, same
// initialize matrixes
fp1( 4, fm1 ); // implicit 6 columns
fp1( 6, fm2 );
```

Variable Dimension

```
#define sub( m, r, c ) *(m + r * sizeof( m[0] ) + c)
void fp2( int rows, int cols, int *m ) {
    ... printf( "%d ", sub( m, r, c ) ); ...
}
int vm1[4][4], vm2[6][8]; // no VLA, different
// initialize matrixes
fp2( 4, 4, &vm1[0][0] );
fp2( 6, 8, &vm2[0][0] );
```

Figure 2.1: Pre-VLA contiguous matrix styles, fixed *vs.* variable

The language decides if the matrix and array-of-array are laid out the same or differently. For example, an array-of-array may be an array of row pointers to arrays of columns, so the rows may not be contiguous in memory nor even the same length (*e.g.*, triangular matrix). This layout occurs in Java's built-in array of built-in array, and in the above C++ vv example. On the other hand, both Pascal declarations and the C++ m declaration give contiguous allocations. Regardless, there is usually a uniform subscripting syntax masking the memory layout, even though a language could differentiate between the two forms using subscript syntax, *e.g.*, m[1,2] *vs.* aa[1][2]. Nevertheless, controlling memory layout can make a difference in what operations are allowed and in performance (*e.g.*, caching/NUMA effects). C also allows non-contiguous arrays-of-arrays:

```
int m[5][5];                    // contiguous
int * aa[5];                    // non-contiguous
```

both with different memory layout using the same subscripting and different degrees of issues.

The focus of this work is on the contiguous multidimensional arrays in C. The reason is that programmers are often forced to use the more complex array-of-array form when a contiguous array would be simpler, faster, and safer. Nevertheless, the C array-of-array form is still important for special circumstances.

Figure 2.2 shows a powerful extension made in C99, for manipulating contiguous *vs.* non-contiguous arrays.[3] For contiguous-array arguments (including VLA), C99 conjoins one or more of the parameters as a downstream dimension(s), *e.g.*, cols, implicitly using this parameter to compute the row stride of m. Hence, if the declaration of fc is changed to:

```
void fc( int rows, int cols, int m[rows][cols] ) ...
```

there is now sufficient information to support array copying and subscript checking to prevent buffer-overflow problems, *but neither feature is provided*. While this contiguous-array capability is a step forward, it is still the programmer's responsibility to manually manage the number of dimensions and their sizes, both at the function definition and call sites. That is, the array does not automatically carry its structure and sizes for use in computing subscripts. While the non-contiguous style in faa looks very similar to fc, the compiler only understands the unknown-sized array of row pointers, and it relies on the programmer to traverse the columns in a row with a

[3]C90 also supported non-contiguous arrays. Though GNU-flavoured C++ eventually got VLAs, it never got this enhancement for managing a multidimensional VLA parameter.

Contiguous

```
void fc( int rows, int cols, int m[ /* rows */ ][cols] ) {
    for ( size_t  r = 0; r < rows; r += 1 ) {
        for ( size_t  c = 0; c < cols; c += 1 )
            ... m[r][c] ...
            // each r−step: cols * sizeof(int)
}
int m[5][5];
for ( int r = 0; r < 5; r += 1 ) {

    for ( int c = 0; c < 5; c += 1 )
        m[r][c] = r + c;
}
fc( 5, 5, m );
```

Non-contiguous

```
void faa( int rows, int cols, int * m[ /* cols */ ] ) {
    for ( size_t  r = 0; r < rows; r += 1 ) {
        for ( size_t  c = 0; c < cols; c += 1 )
            ... m[r][c] ...
            // each r−step: 1 * sizeof(int*)
}
int * aa[5];  // row pointers
for ( int r = 0; r < 5; r += 1 ) {
    aa[r] = malloc( 5 * sizeof(int) ); // create rows
    for ( int c = 0; c < 5; c += 1 )
        aa[r][c] = r + c;
}
faa( 5, 5, aa );
```

Figure 2.2: C99 matrix styles, contiguous *vs.* non-contiguous

correctly bounded loop index. Specifically, there is no requirement that the rows are the same length, like for a poem with different-length lines.

2.3.4 Multi-Dimensional Arrays Decay and Pointers Diffract

As for single-dimension, multi-dimensional arrays have similar issues (see Section 2.3, p. 10). Again, the inspection begins by using **sizeof** to provide program semantics for the intuition of an expression's type.

```
float mx[3][10];
static_assert( sizeof(float) == 4 );          // floats (atomic elements) are 4 bytes
static_assert( sizeof(void *) == 8 );         // pointers are 8 bytes
```

There are now three means of deriving expressions from mx: *itself*, *first element*, and *first grand-element* (meaning, first element of first element).

```
static_assert( sizeof(mx) == 120 );           // the array, float[3][10]
static_assert( sizeof(mx[0]) == 40 );         // its first element, float[10]
static_assert( sizeof(mx[0][0]) == 4 );       // its first grand element, float

static_assert( sizeof(&(mx)) == 8 );          // pointer to the array, float(*)[3][10]
static_assert( sizeof(&(mx[0])) == 8 );       // pointer to its first element, float(*)[10]
static_assert( sizeof(&(mx[0][0])) == 8 );    // pointer to its first grand-element, float*
```

Given that arrays are contiguous and the size of **float** is 4, then the size of mx with 3×10 floats is 120 bytes, the size of its first element (row) is 40 bytes, and the size of the first element of the first row is 4. Again, an array and a pointer to anything it contains are different.

```
float (*pm)[3][10] = &(mx);
float (*pm0)[10] = &(mx[0]);
float *pm00 = &(mx[0][0]);
```

```
static_assert( (void *)&mx == (void *)&(mx[0] ) );
static_assert( (void *)&mx == (void *)&(mx[0][0]) );

assert( (void *) pm == (void *) pm0 );
assert( (void *) pm == (void *) pm00 );
```

As well, there is pointer decay from each of the matrix axes to pointers, all having the same address. Finally, subscripting is allowed on a malloc result, where the referent may or may not allow subscripting or have the right number of subscripts.

2.3.5 Array Parameter Declaration

Passing an array as an argument to a function is necessary. Assume a parameter is an array where the function intends to subscript it. This section asserts that a more satisfactory/formal characterization does not exist in C, then surveys the ways that C API authors communicate that p has zero or more dimensions, and finally calls out the minority cases where the C type system is using or verifying such claims.

A C parameter declaration looks different from the caller's and callee's perspectives. Both perspectives consist of the text read by a programmer and the semantics enforced by the type system. The caller's perspective is available from a function declaration, which allows definition-before-use and separate compilation, but can also be read from (the non-body part of) a function definition. The callee's perspective is what is available inside the function.

```
int foo( int, float, char );                // declaration, parameter names optional
int bar( int i, float f, char c ) {         // definition, parameter names mandatory
    // callee's perspective of foo and bar
}
// caller's perspectives of foo and bar
```

From the caller's perspective, the parameter names are semantically optional, but provide useful comments. From the callee's perspective, parameter names are semantically essential. Array parameters introduce a further, subtle, semantic difference and considerable freedom to comment.

At the semantic level, there is no such thing as an array parameter, except for one case, T [**static** 5], discussed shortly. Instead, there are only pointer parameters. This fact probably shares considerable responsibility for the common sense of *an array is just a pointer*, which has been refuted in non-parameter contexts. This fact holds in both the caller's and callee's perspectives. However, a parameter's type can include "array of", *e.g.*, the type "pointer to array of 5 ints" (T (*)[5]), which is a pointer type. This type is fully meaningful in the sense that its description does not contain any information that the type system ignores, and the type appears the same in the caller's and callee's perspectives. In fact, the outermost type constructor (syntactically first dimension) is really the one that determines the parameter flavour.

C largely ignores length information given in parameter declarations, when determining a function's type. For example, it accepts these assignments between the f- and g-types,

```
float sum( float a[5] );      float sum( float a[] );      float sum( float *a );       // array of float
float sum( float m[5][4] );   float sum( float m[][4] );   float sum( float (*m)[4] );   // matrix of float
float sum( float a[5][] );    float sum( float a[][] );    float sum( float (*a)[] );   // invalid
float sum( float a[5]* );     float sum( float a[]* );     float sum( float (*a)* );    // invalid
float sum( float *a[5] );     float sum( float *a[] );     float sum( float **a );      // array of ptr to float
```

Figure 2.3: Multiple ways to declare an array parameter. Across a valid row, every declaration is equivalent. Each column gives a declaration style, where the style for that column is read from the first row. The second row begins the style for multiple dimensions, with the rows thereafter providing context for the choice of which second-row [] receives the column-style variation.

```
void f( int, float[][42] );          // array of len-42 arrays
void g( int n, float[][n]  );        // array of VLAs

typeof( & f ) fp = & g;
typeof( & g ) gp = & f;
```

while entirely different stride calculations would occur in the function bodies of f and g.

Yet, C allows array syntax for the outermost type constructor, from which comes the freedom to comment. An array parameter declaration can specify the outermost dimension with a dimension value, [10] (which is ignored), an empty dimension list, [ ], or a pointer, *, as seen in Figure 2.3. Examining the rows, the rationale for rejecting the first invalid row (row 3) follows shortly, while the second invalid row (row 4) is nonsense, included to complete the pattern; its syntax hints at what the final row actually achieves. Note that in the leftmost style, the typechecker ignores the actual value, even for a dynamic expression.

```
int N;
void foo( float a[N] ) ; // N is ignored
```

It is a matter of taste as to whether a programmer should use the left form to get the most out of commenting subscripting and dimension sizes, sticking to the right (avoiding false comfort from suggesting the typechecker is checking more than it is), or compromising in the middle (reducing unchecked information, yet clearly stating, “I am for subscripting”).

Note that this equivalence of pointer and array declarations is special to parameters. It does not apply to local variables, where true array declarations are possible (formal *vs.* actual declarations).

```
void f( float * a ) {
    float * b = a;      // ok
    float c[] = a;      // reject
    float d[] = { 1.0, 2.0, 3.0 };    // ok
    static_assert( sizeof(b) == sizeof(float*) );
    static_assert( sizeof(d) != sizeof(float*) );
}
```

The consequence of this equivalence is that the type system does not help a caller get it right.

```
float sum( float v[] );
float arg = 3.14;
sum( &arg );                         // accepted, v = &arg
```

Given the syntactic dimension forms [ ] or [5], it raises the question of how to qualify the implied array pointer rather than the array element type. For example, the qualifiers after the * apply to the array pointer.

```
void foo( const volatile int * const volatile );
void foo( const volatile int [ ] const volatile ); // does not parse
```

C instead puts these pointer qualifiers syntactically into the first dimension.

[*type-qualifier-list*$_{opt}$ *assignment-expression*$_{opt}$]

```
void foo( int [const  volatile] );
void foo( int [const  volatile 5] );            // 5 is ignored
```

To make the first dimension size meaningful, C adds this form.

[**static** *type-qualifier-list*$_{opt}$ *assignment-expression*]

```
void foo( int [static 3] );
int ar[10];
foo( ar );        // check argument dimension 10 > 3
```

Here, the **static** storage qualifier defines the minimum array size for its argument. Earlier versions of GCC (< 11) and possibly clang ignore this dimension qualifier, while later versions implement the check, in accordance with the standard.

Note that there are now two different meanings for modifiers in the same position. In

```
void foo( int x[static const volatile 3] );
```

the **static** applies to the 3, while the **const volatile** applies to the x.

With multidimensional arrays, on dimensions after the first, a size is required, and is not ignored. These sizes (strides) are required for the callee to subscript.

```
void f( float a[][10], float b[][100] ) {
    static_assert( ((char*)&a([1])) - ((char*)&a([0])) == 10 * sizeof(float) );
    static_assert( ((char*)&b([1])) - ((char*)&b([0])) == 100 * sizeof(float) );
}
```

Here, the distance between the first and second elements of each array depends on the inner dimension size.

The significance of an inner dimension's length is a fact of the callee's perspective. In the caller's perspective, the type system is quite lax. Here, there is (some, but) little checking if the argument being passed matches the parameter.

```
void f( float [][10] );
int n = 100;
float a[100], b[3][12], c[n], d[n][n];
f( a );
f( b );                                 // reject: inner dimension 12 for 10
f( c );
f( d );                                 // accept with inner dimension n for 10
f( &a );                                // reject: inner dimension 100 for 10
f( &b );
f( &c );                                // accept with inner dimension n for 10
f( &d );
```

The cases without comments are rejections, but simply because the array ranks do not match; in the commented cases, the ranks match and the rules are discussed apply. The cases f(b) and f(&a) show where some length checking occurs. But this checking misses the cases f(d) and f(&c), allowing the calls with mismatched lengths, actually 100 for 10. The C checking rule avoids false alarms, at the expense of safety, by allowing any combinations that involve dynamic values. Ultimately, an inner dimension's size is a callee's *assumption* because the type system uses declaration details in the callee's perspective that it does not enforce in the caller's perspective.

Finally, to handle higher-dimensional VLAs, C repurposed the * *within* the dimension in a declaration to mean that the callee must make an assumption about the size, but no (unchecked, possibly wrong) information about this assumption is included for the caller-programmer's benefit/-over-confidence.

[*type-qualifier-list*$_{opt}$ *]

```
void foo( float [][*] );                    // declaration
void foo( float a[][10] ) { ... }           // definition
```

Repeating it with the full context of a VLA is useful.

```
void foo( int, float [][*] );               // declaration
void foo( int n, float a[][n] ) { ... }     // definition
```

Omitting the dimension from the declaration is consistent with omitting parameter names, for the declaration case has no name n in scope. The omission is also redacting all information not needed to generate correct caller-side code.

2.3.6 Arrays Could be Values

All arrays have a known runtime size at their point of declaration. Furthermore, C provides an explicit mechanism to pass an array's dimensions to a function. Nevertheless, an array cannot be copied, and hence, not passed by value to a function, even when there is sufficient information to do so.

However, if an array is a structure field (compile-time size), it can be copied and passed by value. For example, a C jmp_buf is an array.

```
typedef long int jmp_buf[8];
```

An instance of this array can be declared as a structure field.

```
struct Jmp_Buf {
    jmp_buf jb;
};
```

Now the array can be copied (and passed by value) because structures can be copied.

```
Jmp_Buf jb1, jb2;
jb1 = jb2;          // copy
void foo( Jmp_Buf );
foo( jb2 );         // copy
```

This same argument applies to returning arrays from functions. There can be sufficient information to return an array by value but it is unsupported. Again, array wrapping allows an array to be returned from a function and copied into a variable.

2.4 Linked List

Linked lists are blocks of storage connected using one or more pointers. The storage block (node) is logically divided into data (user payload) and links (list pointers), where the links are the only component used by the list structure. Since the library does not use the data component, list structures are often polymorphic over the data, which is often homogeneous.

The links organize nodes into a particular shape of data structure, *e.g.*, chain, tree, hash table, *etc.*, with operations specific to that kind. In all these cases, a node's address is significant, so nodes are communicated by reference/pointer, and never by copy (by value).

2.4.1 Design Issues

This thesis focuses on a reduced design space for linked lists that are important to, but not limited to, *system programmers*. Within this restricted space, all design-issue discussions make the following assumptions.

- The chain shape, as opposed to tree or hash table, is considered.
- A doubly-linked list is being designed. Generally, the discussed issues apply similarly for singly-linked lists. Ordered linking (as opposed to circularly-linked) occurs.
- Link fields are system-managed. The system has freedom over how to represent these links. The user works with the system-provided API to query and modify list membership.

Alternatives to these assumptions are discussed under Future Work (Section 4.7, p. 102).

2.4.2 Preexisting Linked-List Libraries

To show examples of the concepts being defined, these two preexisting linked-list libraries are used throughout, and further libraries are introduced as needed.

1. Linux Queue library [43] (LQ) of <sys/queue.h>.
2. C++ Standard Template Library's (STL)[4] std::list [36]

For the discussion, assume the type req (request) is the user's payload in examples. Then the job of a list library is to help a user manage (organize) requests. A request might be a network arrival event processed by a web browser or a thread blocked/scheduled by a runtime.

2.4.3 Link Attachment: Intrusive *vs*. Wrapped

Link attachment deals with the question: Where are the libraries' inter-node link-fields stored, in relation to the user's payload data fields?

Figure 2.4 shows three basic styles. Figure 2.4(a) shows the *intrusive* style, placing the link fields inside the payload structure. Figures 2.4(b) and (c) show the two *wrapped* styles, which place the payload inside a generic library-provided structure that then defines the link fields. The

[4]The term STL is contentious as some people prefer the term standard library.

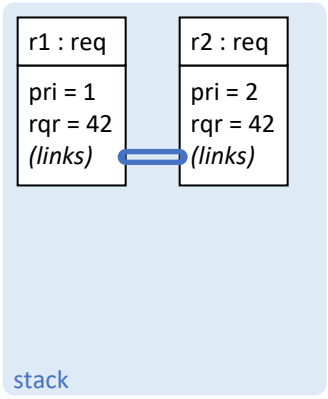
```
// C

struct req {
    int pri, rqr;
    LIST_ENTRY(req) d;
};
LIST_HEAD(reql, req);

struct reql reqs;
LIST_INIT(&reqs);

struct req r1 = {1, 42}, r2 = {2, 42};

LIST_INSERT_HEAD(&reqs, &r2, d);
LIST_INSERT_HEAD(&reqs, &r1, d);
```



(a) Intrusive

```
// C++

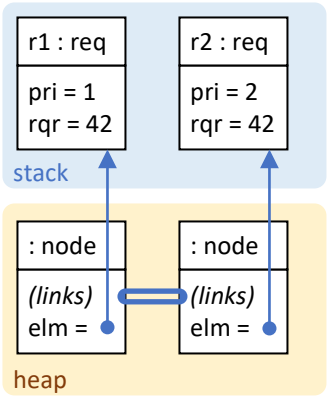
struct req {
    int pri, rqr;

};

list<req *> reqs;

req r1 = {1, 42}, r2 = {2, 42};

reqs.push_front(&r2);
reqs.push_front(&r1);
```



(b) Wrapped reference

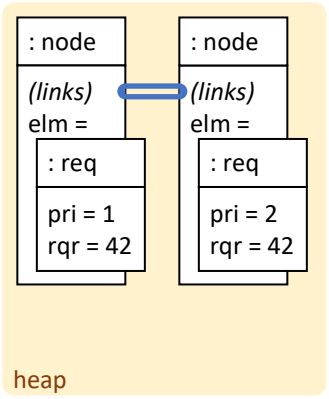
```
// C++

struct req {
    int pri, rqr;

};

list<req> reqs;

reqs.emplace_front(2, 42);
reqs.emplace_front(1, 42);
```



(c) Wrapped value

Figure 2.4: Three styles of link attachment. The diagrams show the memory layouts that result after the code runs, eliding the head object reqs; head objects are discussed in Section 2.4.6, p. 31. In (a), the field req.d names a list axis; these are discussed in Section 2.4.4. In (b) and (c), the type node represents a library-internal type, which is std::_List_node in the GNU implementation (see /usr/include/c++/X/bits/stl_list.h, where X is the g++ version number).

wrapped style distinguishes between wrapping a reference or a value, *e.g.*, list<req *> or list<req>. (For this discussion, list<req &> is similar to list<req *>.) This difference is one of user style, performance (due to copying) and ownership; it is not a matter of framework capability. LQ is intrusive; STL is wrapped, supporting either references or values.

Each diagram in Figure 2.4 uses the fewest dynamic allocations for its respective style: in intrusive, there is no dynamic allocation, in wrapped reference only the linked fields are dynamically allocated, and in wrapped value the copy data-area and linked fields are dynamically allocated. The advantage of intrusive is the control in memory layout and storage placement. Both wrapped styles have independent storage layout and imply library-induced heap allocations, with lifetime that matches the item's membership in the list.

In the LQ column, (a), the macro LIST_INSERT_HEAD(&reqs, &r2, d) takes the list header, a

pointer to the node, and the offset of the link fields in the node. The user's last argument is the field name; the macro expansion uses it as (&r2)->d, causing the compiler to insert the link-field offset. With this information, the API can operate on node pointers, and also work on the link fields within.

An aspect of layout control is allowing the user to specify link fields explicitly, controlling field order and attributes within the req object. LQ allows this ability through the LIST_ENTRY macro[5], which can be placed anywhere in the node. An example of an attribute on the link fields is cache alignment, possibly in conjunction with other req fields, improving locality and/or avoiding false sharing. If a list is frequently traversed in the forward direction, and infrequently gets elements removed at arbitrary positions, then an ideal layout for cache locality puts the forward links, together with frequently-used payload data on one cache line, leaving the reverse links on a colder cache line. In contrast, supplying link fields by inheritance makes them implicit and relies on compiler placement, such as the start or end of req, and no explicit attributes. Wrapped reference has no control over the link fields, but the separate data allows some control; wrapped value has no control over data or links.

Another subtle advantage of intrusive attachment is that a reference to a user-level item (req) is sufficient to navigate or manage the item's membership. In LQ, the intrusive req pointer is the correct argument type for operations LIST_NEXT or LIST_REMOVE; there is no distinguishing a req from a req in a list. The same is not true of STL, wrapped reference or value. There, the analogous operations, iterator::operator++(), iterator::operator*(), and list::erase(iterator), work on a parameter of type list<T>::iterator; there is no mapping from req & to list<req>::iterator.

The advantage of wrapped is the abstraction of a data item from its list membership(s). In the wrapped style, the req type can come from a library that serves many independent uses, which generally have no need for listing. Then, a novel use can put a req in a list, without requiring any upstream change in the req library. In intrusive, the ability to be listed must be planned during the definition of req.

Finally, it is possible to simulate wrapped using intrusive, illustrated in Figure 2.5. This shim layer performs the implicit dynamic allocations that pure intrusion avoids. But there is no reduction going the other way. No shimming can cancel the allocations to which wrapped membership commits.

Because intrusion is a lower-level listing primitive, the system design choice is not between forcing users to use intrusion or wrapping. The choice is whether or not to provide access to an allocation-free layer of functionality. An intrusive-primitive library like LQ lets users choose when to make this tradeoff. A wrapped-primitive library like STL forces users to incur the costs of wrapping, whether or not they access its benefits. Like LQ, the C∀ intrusive library of Chapter 4, p. 79 is capable of supporting a wrapped library, if need arose.

2.4.4 Axis: Single *vs*. Multi-Static *vs*. Dynamic

Axis deals with the question: In how many different lists can a node be stored, at the same time? Figure 2.6, p. 28 shows an example that can traverse all requests in priority order (field pri) or

[5]It is possible to have multiple of these link fields allowing a node to appear on multiple lists simultaneously.

```
template<typename El>
class list {
    struct node {
        LIST_ENTRY(node) links;
        El elem;
    };
    LIST_HEAD(Impl, node);
    Impl impl;
  public:
    list() {
        LIST_INIT(&impl);
    }
    void push_front( const El & src ) {
        node * n = new node();
        n->elem = src;
        LIST_INSERT_HEAD(&impl, n, links);
    }
    // ... `emplace` elided
};
```

Figure 2.5: Simulation of wrapped using intrusive. Illustrated by pseudocode implementation of an STL-compatible API fragment using LQ as the underlying implementation. The gap that makes it pseudocode is that the LQ C macros do not expand to valid C++ when instantiated with template parameters—there is no **struct** El. When using a custom-patched version of LQ to work around this issue, the programs of Figure 2.4(b), p. 25 and wrapped value work with this shim in place of real STL. Their executions lead to the same memory layouts.

navigate among requests with the same requestor (field rqr). Each of "by priority" and "by common requestor" is a separate axis. The example has a single priority-list linked (first axis): [1, 2, 2, 3, 3, 4], where nodes may have the same priority. And it has three common-requestor lists (second axis): [42, 42], [17, 17, 17], and [99], giving four head nodes, one for each list. The example shows an axis can encompass all the nodes (by-priority) or only a subset of the nodes (three by-common-requestor lists).

As stated, the limitation of intrusive is knowing a priori how many groups of links are needed for the maximum number of simultaneous lists. Thus, the intrusive LQ example supports multiple, but statically many, linked lists. It is possible to reuse links for different purposes, *e.g.*, if an item is linked one way at one time and another way at another time, and these times do not overlap, the two different linkings can use the same link fields. This feature is used in the C∀ runtime, where a thread node may be on a blocked or running list, but never on both simultaneously.

Now consider the STL in the wrapped-reference arrangement of Figure 2.4(b), p. 25. Here it is possible to construct the same simultaneity by creating multiple STL lists, each pointing at the appropriate nodes. Each group of intrusive links become the links for each separate STL list. The upside is the unlimited number of lists a node can be associated with simultaneously, as any number of STL lists can be created dynamically. The downside is the dynamic allocation of the link nodes and managing multiple lists. In some cases, it is possible to wrap the multiple lists in another type to hide this implementation issue.

```
struct req {
    int pri, rqr;
    LIST_ENTRY( req ) by_pri;
    LIST_ENTRY( req ) by_rqr;
};

LIST_HEAD( reql, req );

struct reql reqs_pri;
struct reql reqs_rqr_42;
struct reql reqs_rqr_17;
struct reql reqs_rqr_99;

LIST_INIT( &reqs_pri );
LIST_INIT( &reqs_rqr_42 );
LIST_INIT( &reqs_rqr_17 );
LIST_INIT( &reqs_rqr_99 );

struct req
    r42a = {1, 42},
    r42b = {2, 42},
    r17a = {2, 17},
    r17b = {3, 17},
    r17c = {4, 17},
    r99a = {3, 99};

LIST_INSERT_HEAD( &reqs_pri, &r17c, by_pri );
LIST_INSERT_HEAD( &reqs_pri, &r99a, by_pri );
LIST_INSERT_HEAD( &reqs_pri, &r17b, by_pri );
LIST_INSERT_HEAD( &reqs_pri, &r42b, by_pri );
LIST_INSERT_HEAD( &reqs_pri, &r17a, by_pri );
LIST_INSERT_HEAD( &reqs_pri, &r42a, by_pri );

LIST_INSERT_HEAD( &reqs_rqr_42, &r42b, by_rqr );
LIST_INSERT_HEAD( &reqs_rqr_42, &r42a, by_rqr );

LIST_INSERT_HEAD( &reqs_rqr_17, &r17c, by_rqr );
LIST_INSERT_HEAD( &reqs_rqr_17, &r17b, by_rqr );
LIST_INSERT_HEAD( &reqs_rqr_17, &r17a, by_rqr );

LIST_INSERT_HEAD( &reqs_rqr_99, &r99a, by_rqr );
```

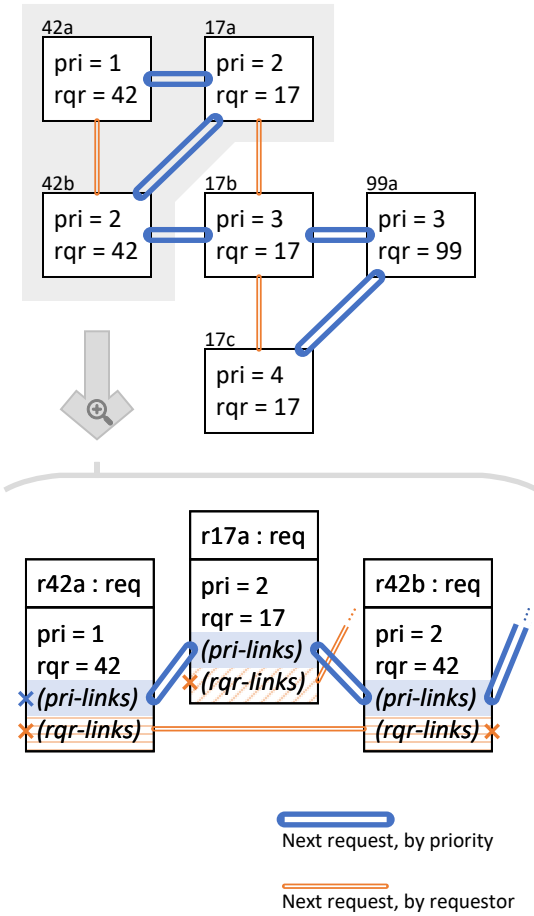


Figure 2.6: Example of simultaneity using LQ lists. The zoomed-out diagram (right/top) shows the complete multi-linked data structure. This structure can navigate all requests in priority order (blue), and navigate among requests with a common request value (orange). The zoomed-in diagram (right/bottom) shows how the link fields connect the nodes on different lists.

Now consider the STL in the wrapped-value arrangement of Figure 2.4(c), p. 25. Again, it is possible to construct the same simultaneity by creating multiple STL lists, each copying the appropriate nodes, where the intrusive links become the links for each separate STL list. The upside is the same as for wrapped-reference arrangement with an unlimited number of list bindings. The downside is the dynamic allocation, significant storage usage, and cost of copying nodes. As well, it is unclear how node updates work in this scenario, without some notation of ultimately merging node information.

An alternative system providing intrusive data-structures is μC++, a concurrent extension of C++. It provides a basic set of intrusive lists [8, appx. F], where the link fields are defined with the data fields using inheritance.

singly-linked list

```
struct Req : public uColable {
    int pri, rqr;
    Req( int p, int q ) : pri{p}, rqr{q} {}
};
```

doubly-linked list

```
struct Req : public uSeqable {
    int pri, rqr;
    Req( int p, int q ) : pri{p}, rqr{q} {}
};
```

A node can be placed in the following data structures depending on its link fields: uStack and uQueue (singly linked), and uSequence (doubly linked). A node inheriting from uSeqable can appear in a singly or doubly linked structure. Structure operations implicitly know the link-field location through the inheritance.

```
uStack<Req> stack;
Req r;
stack.push( r );  // link fields at beginning of node
```

Simultaneous axes cannot be done with C++ multiple inheritance, because there is no mechanism either to know the order of inheritance fields or to name each inheritance. Instead, a special type is required that contains the link fields and points outward at the node.

```
struct Req : public uColable {
    int pri, rqr;
    ByRqr by_rqr;  // second set of links
    Req( int p, int q ) : pri{p}, rqr{q}, by_rqr{*this} {}
};
```

```
struct ByRqr : public uSeqable {
    Req & outer;  // node pointer
    ByRqr( Req & outer ) : outer( outer ) {}
    Req & get() const { return outer; }
};
```

This node can now be inserted into a subsequent doubly-linked list through these explicitly embedded intrusive links.

```
uSequence<ByRqr> reqs_rqr_42;
reqs_rqr_42.add_front( r.by_rqr );          // link fields in embedded type
ByRqr node_inr = reqs_rqr_42.remove( r.by_rqr );
int rp = node_inr.get().pri;                // indirection to node
```

Hence, the μC++ approach streamlines one set of intrusive links through the C++ inheritance mechanism, and falls back onto the LQ approach of explicit declarations for additional intrusive links. However, μC++ does not have the LQ trick of passing a field name as a macro argument, resulting in the somewhat more cumbersome realization of subsequent link fields.

The major ergonomic difference among the approaches is naming and name usage. The intrusive model requires naming each set of intrusive links, *e.g.*, by_pri and by_rqr in Figure 2.6. μC++

cheats by using inheritance for the first intrusive links, after which explicit naming of intrusive links is required. Furthermore, these link names must be used in all list operations, including iterating, whereas wrapped reference and value hide the list names in the implicit dynamically-allocated node-structure.

One issue here is whether an API for simultaneity can support one *implicit* list, when several are not wanted. μC++ supports it, LQ does not, and STL does not have this question.

The other issue here is whether the API's subsequent-axis scheme avoids extra[6] user effort or runtime steps. A μC++ subsequent axis incurs: defining an application-specific boilerplate type (ByRqr), referring to its field (.by_rqr) in sequence operations where the sequence's type (uSequence< ByRqr >) already implies the axis[7] and incurs both code-bloat and runtime pointer indirection at node_inr.get().pri. LQ avoids extras, μC++ does not, and STL does not have this question.

2.4.5 User Integration: Preprocessed *vs*. Type-System Mediated

While the syntax for LQ is reasonably succinct, it comes at the cost of using C preprocessor macros for generics; macros are not part of the language's type system. Hence, small errors in macro arguments can lead to large substitution mistakes, as the arguments may be textually written in many places and/or concatenated with other arguments/text to create new names and expressions. Hence, textual expansion can lead to a cascade of error messages that are confusing and difficult to debug. For example, argument errors like a.b,c, comma instead of period, or by-pri, minus instead of underscore, can produce many error messages. Similar problems exist for C++ templates.

Further issues with macros occur when the substitution result contains artifacts related to evaluation order that are not evident in the original. A real problem that occurred while preparing the linked-list performance evaluation stemmed from the innocent-looking

```
TAILQ_REMOVE(reqs, TAILQ_LAST(reqs, reql), d);
```

not being equivalent to the explicitly multi-step version:

```
struct req * last = TAILQ_LAST(reqs, reql);
TAILQ_REMOVE(reqs, last, d);
```

It turns out that TAILQ_REMOVE uses its "which element to remove" parameter at several places, including, importantly, one occurring after the removal's changes are in progress. When the second use encounters the macro substitution TAILQ_LAST(reqs, reql), it obtains a different node than the first use got, with the removal's changes having already started. This macro-induced phenomenon led to an invalid pointer dereference (safety violation), at a run-time well after the removal at issue (costly to resolve [48]).

Instead, language function calls (even with inlining) handle argument mistakes locally at the call, giving a very specific error message. In the world of C++ templates, concepts were introduced

[6]No objective baseline for "extra" is stated; the obvious candidates are rejected in the interest of letting the paragraph conclude with synthetic statements. Rather, an individual case's evaluation appeals to the reader's sense of reasonable software engineering.

[7]A hypothetical alternative formulation, where the application-specific ByRqr is replaced by a framework-provided uSeqableSubs< Req >, would (anul the first limitation and) reformulate this second limitation as enabling the user to mismatch axes, as in, reqs_rqr_42.remove(r.by_cost) being well-typed.

to deal with this problem. Furthermore, language function calls use (only) C-language semantics for argument evaluation, instead of appealing to reasoning about how the semantics might play out in an invisible substitution result.

So, avoiding a macro-centric implementation helps users avoid and diagnose critical mistakes.

2.4.6 List Identity: Headed *vs*. Ad-Hoc

All examples so far use two distinct types for: an item found in a list (type req of variable r1, in Figure 2.4, p. 25), and the list "itself" (type reql of variable reqs in the same example, or reqs_pri, in Figure 2.7). This kind of list is *headed*, where the empty list is just a head. An alternate *ad-hoc* approach omits the header. Here, all link traversals begin from an existing reference to a node. A given node may be able to see that it is first, but there is no direct access to a global first. A headed-list API is a superset of an ad-hoc list's, and can normally perform all of the ad-hoc operations. Figure 2.7 shows both approaches for different list lengths and unlisted elements. For headed, there are length-zero lists (heads with no elements), and an element can be listed or not listed. For ad-hoc, there are no length-zero lists[8] and every element belongs to a list of length at least one.

The purpose of a header is to provide specialized but implicit node access, such as the first/last nodes in the list, where accessing these nodes is deemed a commonly occurring operation and should be $O(1)$ for performance of certain operations. For example, without a last pointer in a singly-linked list, adding to the end of the list is an $O(N)$ operation to traverse the list to find the last node. Without the header, this specialized information must be managed explicitly, where the programmer builds their own external, equivalent header information. However, external management of particular nodes might not be beneficial because the list does not provide operations that can take advantage of them, such as using an external pointer to update an internal link. Clearly, there is a cost maintaining this specialized information, which needs to be amortized across the list operations that use it, *e.g.*, rarely adding to the end of a list. A runtime component of this cost is evident in LQ's offering the choice of type generators LIST *vs*. TAILQ. Its LIST maintains a *first*, but not a *last*; its TAILQ maintains both roles. (Both types are doubly linked and an analogous choice is available for singly linked.)

Libraries have not traditionally offered an ad-hoc list abstraction, but it is a common pattern when programmers roll their own inter-linked type. The ad-hoc pattern obviates the questions, "Who should own the head?" or, "Are we sure the head outlives the elements?" Providing library support for ad-hoc lists is an opportunity to lower the incentive for rolling one's own.

2.4.7 End Treatment: Uniform *vs*. Cased

All lists must have a logical *beginning/ending*, otherwise list traversal is infinite. *End treatment* refers to how the list represents the lack of a predecessor/successor at the ends. For example, a doubly-linked list might set its last node's successor-pointer to null. As the following discussion establishes, this choice can have consequences in performance and API complexity.

[8] Up to philosophical fun about nothingness occurring everywhere. If a pattern of memory reads leads to the conclusion, "Zero elements there, but it just as well could have been twelve," then this pattern is a head.

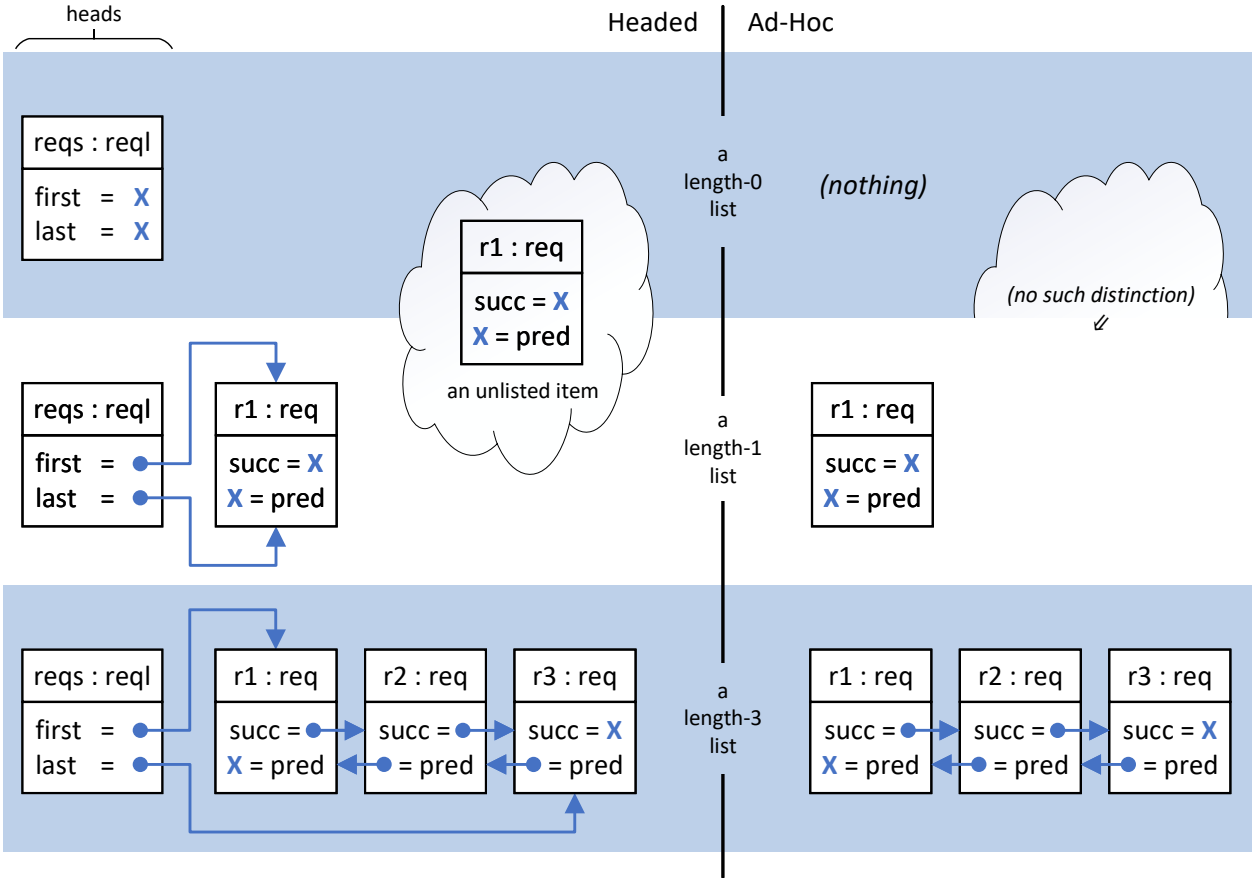


Figure 2.7: Comparison of headed and ad-hoc list identities, for various list lengths. Pointers are logical, meaning that no claim is intended about which part of an object is being targeted.

The LQ list representation is detailed in Figure 2.8. This figure shows a headed list with three user items. LQ uses a null succ pointer on the last element. But it uses a trick for managing the pred pointer, with which even the first node has a non-null pred. LQ demonstrates both end treatments via the different ways it handles forward and reverse linking.

A *uniform* end treatment occurs when list operations can use the same logic on an end element as they use on a middle element. A *cased* end treatment occurs when different logic is required for end *vs.* middle.

Figure 2.9, p. 34 gives a timeline that traces LQ operation executions. The conclusion is that LQ has uniform end treatment at its "first" end, and cased end treatment at its "last" end.

In the figure's topmost step, the initial state matches that of Figure 2.8, now with the three user items named a, b and c. Four list API calls then follow sequentially, each adding an item named w, x, y or z at the original list's four distinct insertion-position candidates.

The first two calls are working in the succ/last direction. The very first is working on a middle position; there, no null succ pointer is encountered. The second is working on an end position; there, a null succ pointer arises. Attempting to traverse (listelm)–>succ–>pred, as in the first case, would yield an error in the second case, so an alternate handling is needed. In the LQ design,

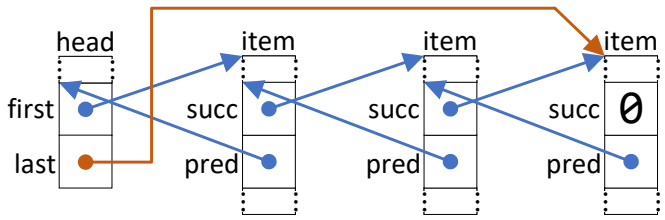


Figure 2.8: LQ field-level representation of links and ends. Each object's memory is pictured as a vertical strip. Each arrow points at a significant location within a strip. Uniform treatment of the first-end is evident from the assertion (**this->pred == this) holding for all nodes this, including the first one. Cased treatment of the last-end is evident from the last-end analogous proposition, (this->succ->pred == &this->succ), is false when this is the last node.

this handling is to work on the head, assigning (head)->last, in lieu of managing "the successor at 0x0." In both of the calls, an is-null check occurs, to distinguish the two cases. Due to this check guarding two *different* handlings, the last-end treatment is cased.

The final two calls are working in the pred/first direction. In this direction, LQ's clever trick applies. Am LQ pred is not a pointer to the previous *node*. Rather, it points at the current node's incoming *forward pointer*. So, when the third call works on its (before-b) middle position, chasing b->pred gives &(a->succ) as b's incoming forward pointer. And when the last call works on its (before-a) end position, chasing a->pred gives &(lst->first) as a's incoming forward pointer. In both cases, the correct action for achieving the desired insertion is to adjust this found incoming forward pointer, setting it to reference the item being inserted. Due to the *same* logic applying for the last two calls, the first-end treatment is uniform.

An operation like LQ's insert-after contains a branch to decide which case is happening. All branches have pathological cases where branch prediction's success rate is low and the execution pipeline stalls often. Hence, the end-treatment issue is relevant to achieving high performance.

Moreover, LQ's insert-after needs extra data to handle the end-position case: direct access to the list head. Insert-before, on the other hand, simply does not have a head parameter. Logically, the head information is redundant; providing a before/after reference position says unambiguously where the new item should go. But at the LQ "last" end, LQ's null-succ *implementation* choice makes it demand an extra input that is used only in an edge case. Redundant inputs are opportunities for user error and edge cases are prone to going unexercised during testing. Allowing cased end treatment to leak into a list API invites poor quality.

2.5 String

A string is a sequence of symbols, where the form of a symbol can vary significantly. When drawn for human reading, text can be left-to-right, right-to-left, top-to-bottom, or have stacked elements (*e.g.*, Arabic). But a string serves all of these presentations by forming the sequence of symbols that matches the order in which the text is read.

A C character constant is enclosed in single-quotes, *e.g.*, `'x'`, `'£'`. C character constants have type **int**, but with conversion to **char** available implicitly. A character can be formed from an

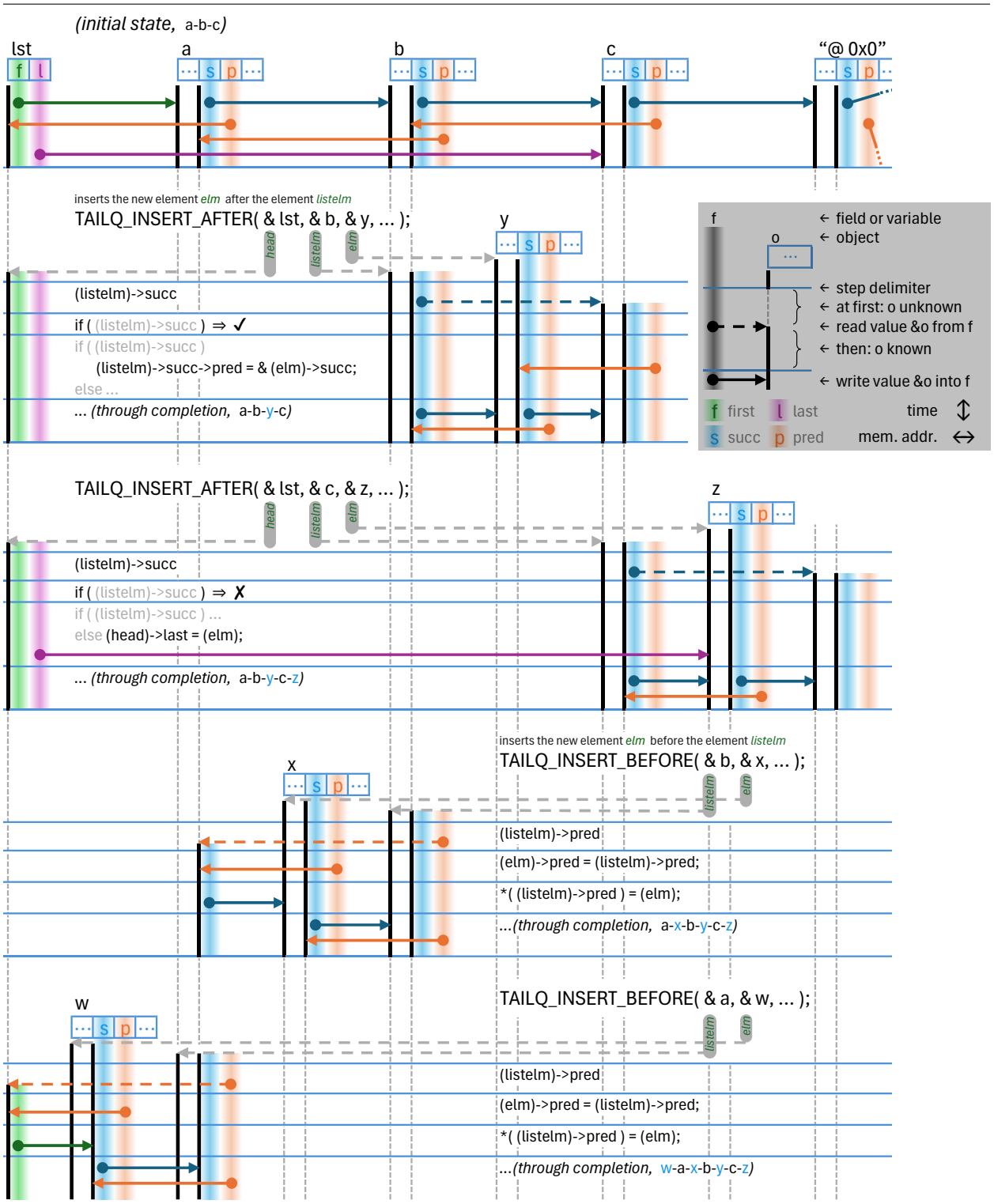


Figure 2.9: Element-relative insertion operations on LQ's uniform-first cased-last representation.

escape sequence, which expresses a non-typable character, *e.g.*, `'\f'` for form-feed or `'\''` for embedded single quote. Escape sequences also encode raw numeric values; *e.g.*, `'\x61'` gives the 8-bit letter "a."

A C string constant is zero or more regular or escape characters enclosed in double-quotes `"xyz\n"`. A C string is an array whose elements are single characters.

C strings are null-terminated rather than maintaining a separate string length.

> The compiler automatically places the null character \0 at the end of each such string, so programs can conveniently find the end. This representation means that there is no real limit to how long a string can be, but programs have to scan one completely to determine its length. [32, p. 36]

This property is only preserved by the compiler with respect to character constants, *e.g.*, `"abc"` is actually {`'a'`, `'b'`, `'c'`, `'\0'`}, *i.e.*, 4 characters rather than 3. Otherwise, the compiler does not participate, making string operations both unsafe and inefficient. For example, it is common in C to forget:

- that a character-array constant is larger than it appears during manipulation,
- that extra storage is needed in a character array for the terminator, or
- that the terminator must be preserved during string operations, otherwise there are array overruns.

Finally, the need to scan an entire string repeatedly to determine its length can result in significant cost, as it is impossible to cache the length in many cases, *e.g.*, when a string is passed into another function.

C strings are fixed size, because arrays are used for the implementation. However, string manipulation commonly results in dynamically-sized temporary and final string values, *e.g.*, strcpy, strcat, strcmp, strlen, strstr, *etc*. As a result, storage management for C strings is a nightmare, quickly resulting in array overruns and incorrect results.

Collectively, these design decisions make working with strings in C awkward, time consuming and unsafe. While there are companion string routines that take the maximum lengths of strings to prevent array overruns, *e.g.*, strncpy, strncat, strncpy, that means the semantics of the operation can fail because strings are truncated. Suffice it to say, C is not a go-to language for string applications, which is why C++ introduced the dynamically-sized string type.

2.5.1 Character types

While a string is a sequence of characters in computer memory, many interpretations of "a character" are possible. This thesis's string contribution leaves proper treatment of a globally sufficient alphabet as future work (see Section 5.5, p. 141). To enable later precision about what works and what does not, this section describes the contemporarily correct treatment of characters.

The space of available characters and how they are built can be given with different degrees of technical abstraction and at different granularities. It is helpful to have a standard terminology; that of [56] follows, as summarized in [28].

A *character set* defines an alphabet of characters, including whitespace units and control codes. A *coded character set* enumerates a character set, associating each with a numeric value. Two characters sets are relevant to string work in C∀, though a history of computer programming features many more. ASCII [10] defines 128 characters, while Unicode [2] defines about 1.1 M valid numeric codes [2, §2.4], of which about 300 k [54] are currently assigned. An ASCII character's number always fits in a single byte and a Unicode's always fits in four. Neither scheme uses all numbers that this byte count affords. Unicode defines its first 128 characters identically to ASCII's.

An *encoding* associates character-set-valid numbers with binary patterns. ASCII has only the trivial encoding: every character-number is stored as the equivalent unsigned single-byte integer. Unicode's trivial encoding is UTF-32, in which every character uses four bytes. Because texts that use limited alphabets are common, Unicode also has encondings that use bytes more frugally. UTF-8 and UTF-16 are *variable-width* encodings that use one one or two bytes, respectively, in their best cases, and four bytes in their worst cases. ASCII and UTF-32 are *fixed width*. Fixed-width encodings have the advantage of fast random access to the i^{th} character of a string, variable-width encodings require a linear scan through the string for this operation. Among the Unicode variable encodings, UTF-8 offers the advantage that the first 128 Unicode characters have the same binary representations as the ASCII characters that they are defined to match. So, all ASCII text is also UTF-8 text, with identical meaning. Notably, the null-terminating zero character is the sole meaning of a zero byte in UTF-8, while UTF-16 encodes English text with every other byte being zero, and a null terminator requires an aligned double-zero. The UTF-16 advantage is that it is practically fixed-width, under the assumption that the alphabet is restricted to that of a technical manual in a currently living language. This advantage is somewhat historical, in that UTF-16 grew out of an encoding for an older Unicode version that covered precisely this case. More contemporarily, the standard allows for Asian regional slang characters and emojis, which are prevalent in natural text; these take four bytes in both UTF-8 and -16. UTF-16 remains a significant encoding because the early multi-language adopters, Java and Windows, continue base their text support on it.

In C, wider character constants are the same as byte character constants, except prefixed by a width-modifying prefix code, among:

1. `u8'a'` for 1-byte type `char8_t` (since C23),
2. `u'a'` for 2-byte type `char16_t`,
3. `U'a'` for 4-byte type `char32_t`,
4. `L'a'` for platform-defined type `wchar_t`, which is typically 4-byte on Linux and 2-byte on Windows.

Higher-width escape sequences also encode more raw numeric values. For example, while `'\x61'` gives the 8-bit letter "a," `U'\x61'`, `U'\u0061'` and `U'\U00000061'` all give its 32-bit equivalent.

In all cases, character printing is external to C; C consumes/produces bytes of input/output. With familiar standard-out printing, the three-byte sequence `'\xe2'-'\xa9'-'\xbd'` shows "â©½" on a Latin-1 terminal[9] and "≤" on a UTF-8 terminal.

[9]Latin-1 is a contemporarily obscure character set, in which all 256 8-bit values are mapped, providing coverage of many European languages. Like Unicode, the first 128 characters match ASCII. The byte values in the example are not valid in ASCII.

In a string literal, the character type is controlled by a prefix: when prefixed by the codes `L`, `u`, `U`, or `u8`, the array elements have the type of the corresponding character-literal prefix; with no prefix, the element type is **char**. This prefix choice further affects character encoding, a decision not applicable for a character literal. Prefix `u8` gives UTF-8; `u` gives UTF-16; `U` gives UTF-32; `L` picks either `u` or `U`, as it would for a character literal; no prefix picks the *execution character set*, which GCC controls by the `-fexec-charset` feature flag, and which defaults to UTF-8 on Linux. By contrast, a character literal always encodes the numeric value directly. So, `u'\u2a7d'` is a 16-bit value, `u"\u2a7d"` is an array whose first (16-bit) entry is the same value, `u8'\u2a7d'` is illegal (because the requested number does not fit in 16 bits), and `u8"\u2a7d"` is an array with the character's value shredded into three 8-bit entries according to UTF-8 encoding, the same array as `u8"\xe2\xa9\xbd"`. With everything set to UTF-8, printing `"a \xe2\xa9\xbd b"` and `"a \u2a7d b"` both write the same bytes, which show as "a ⩽ b."

An *input character set* allows for writing such strings in source-code files more naturally. GCC's `-finput-charset` specifies which to use, with the default on Linux being, again, UTF-8. UTF-8, -16 and -32 can all express the "⩽" character; so the string literal `"a ⩽ b"` is acceptable in any of these encodings, though each represents it by a different source-file byte pattern.

This association of text "⩽" with byte pattern `"\xe2\xa9\xbd"` is a phenomenon of (UTF-8) string literals, never of character literals. A one-byte `'-'` or `u8'-'` (resp. two-byte `u'-'`) C character cannot represent a Unicode character beyond the 128 ASCII-overlapping range (resp. approx. 33 k technical-manual range). So, even with a suitable input character set, `'ℝ'`[10], `u8'ℝ'` and `u'ℝ'` are all illegal because this character's Unicode code does not fit in 8/16 bits. For alphabets that exceed the trivial-encoding overlap ranges, C character literals are inappropriate. Rather, common practice for variable-width encodings is to use *strings* ubiquitously, even for known-length-one cases. This approach pays a null-terminator tax, including requiring programmers to think **char**[5] or longer for a UTF-8 character, wchar[3] for UTF-16.

In the Linux-style UTF-8 world, much Unicode "support" comes for free. A programmer needs to make the "string is the new character" shift just described. Upon doing so, "bytes in, bytes out" covers considerably much usage.

[10] Unless the execution character set is UTF-32.

Chapter 3

Array

Arrays in C are possibly the single most misunderstood and incorrectly used feature in the language (see Section 2.3, p. 10), resulting in the largest proportion of runtime errors and security violations. This chapter describes the new C∀ language and library features that introduce a length-checked array type, array, to the C∀ standard library [11].

Offering the array type, as a distinct alternative to the C array, is consistent with C∀'s goal of backwards compatibility, *i.e.*, virtually all existing C (gcc) programs can be compiled by C∀ with only a small number of changes, similar to C++ (g++). However, a few compatibility-breaking changes to the behaviour of the C array are necessary, both as an implementation convenience and to fix C's lax treatment of arrays. Hence, the array type is an opportunity to start from a clean slate and show a cohesive selection of features, making it unnecessary to deal with every inherited complexity of the C array. Future work is to sugar the array syntax to look like C arrays (see Section 3.9, p. 74).

3.1 Introduction

The new C∀ array is declared by instantiating the generic array type, much like instantiating any other standard-library generic type (such as C++ vector), using a new style of generic parameter.

```
array( float, 99 ) x;                    // x contains 99 floats
```

Here, the arguments to the array type are **float** (element type) and 99 (dimension). When this type is used as a function parameter, the type system requires the argument to be a perfect match.

```
void f( array( float, 42 ) & p ) {}      // p accepts 42 floats
f( x );                                  // statically rejected: type lengths are different, 99 != 42

test2.cfa:3:1 error: Invalid application of existing declaration(s) in expression.
Applying untyped:  Name: f ... to:  Name: x
```

Function f's parameter expects an array with dimension 42, but the argument dimension 99 does not match.

A function can be polymorphic over array arguments using the C∀ **forall** declaration prefix.

```
forall( T, [N] )
void g( array( T, N ) & p, int i ) {
    T elem = p[i];                  // dynamically checked: requires 0 <= i < N
}
g( x, 0 );                          // T is float, N is 99, dynamic subscript check succeeds
g( x, 1000 );                       // T is float, N is 99, dynamic subscript check fails
```

```
Cforall Runtime error: subscript 1000 exceeds dimension range [0,99) for array 0x555555558020.
```

Function g takes an arbitrary type parameter T and an unsigned integer *dimension* N. The dimension represents a to-be-determined number of elements, managed by the type system, where 0 represents an empty array. The type system implicitly infers **float** for T and 99 for N. Furthermore, the runtime subscript x[0] (parameter i being 0) in g is valid because 0 is in the dimension range $[0, 99)$ for argument x. The call g(x, 1000) is also accepted at compile time. However, the subscript, x[1000], generates a runtime error, because 1000 is outside the dimension range $[0, 99)$ of argument x. In general, the **forall**(..., [N]) participates in the user-relevant declaration of the name N, which becomes usable in parameter/return declarations and within a function. The syntactic form is chosen to parallel other **forall** forms:

```
forall( [N] ) ...                   // dimension
forall( T ) ...                     // value datatype
forall( T & ) ...                   // opaque datatype
```

The generic array type is comparable to the C array type, which C∀ inherits from C. Their runtime characteristics are often identical, and some features are available in both. For example, assume a caller has an argument that instantiates N with 42.

```
forall( [N] )
void f( ... ) {
    float x1[N];                    // C array, no subscript checking
    array(float, N) x2;             // C∀ array, subscript checking
}
```

Both of the variables, x1 and x2, name 42 contiguous, stack-allocated **float** values. The two variables have identical size and layout, with no additional "bookkeeping" allocations or headers. The C array, x1, has no subscript checking, while C∀ array, x2, does. This explicit, generic approach requires a significant extension to the C∀ type system to support a full-featured, safe, efficient (space and time) array type, which forms the foundation for more complex array forms in C∀. In all following discussion, "C array" means types like x1 and "C∀ array" means types like x2.

A future goal is to provide the new array features with syntax approaching C's (declaration style of x1). Then, the library array type could be removed, giving C∀ a largely uniform array type. At present, C∀ array features are only partially supported through C-style syntax, so the generic array is used exclusively in the thesis.

My contributions in this chapter are:

1. A type system enhancement that lets polymorphic functions and generic types be parameterized by a numeric value: **forall**([N]).
2. Provide a dimension/subscript-checked array type in the C∀ standard library, where the array's length is statically managed and dynamically valued.

3. Provide argument/parameter passing safety for arrays and subscript safety.
4. Identify the interesting abilities available by the new array type.

3.2 Features Added

This section shows more about using the C∀ array and dimension parameters, demonstrating syntax and semantics by way of motivating examples. As stated, the core capability of the C∀ array is tracking all dimensions within the type system, where dynamic dimensions are represented using type variables.

By declaring type variables at the front of object declarations, an array dimension is lexically referenceable where it is needed. For example, a declaration can share one length, N, among a pair of parameters and return type, meaning the input arrays and return array are the same length.

```
forall( [N] )                                   // array dimension
array( bool, N ) & f( array( float, N ) & x, array( float, N ) & y ) {
    array( bool, N ) & ret = *alloc();          // sizeof ret used by alloc
    for ( i; N ) {
        ret[i] = 0.005 > 2 * (abs( x[i] − y[i] )) / (abs( x[i]) + abs(y[i] ));
    }
    return ret;
}
```

Function f does a pointwise comparison of its two input arrays, checking if each pair of numbers is within half a percent of each other, returning the answers in a newly allocated **bool** array. The dynamic allocation of the ret array uses the library alloc function,

```
forall( T & | sized(T) )
T * alloc() {
    return (T *)malloc( sizeof(T) );   // C malloc
}
```

which captures the parameterized dimension information implicitly within its **sizeof** determination, and casts the return type. Note that alloc only sees the whole type for its T, array(**bool**, N), where this type's size is a computation based on N. This example illustrates how the new array type plugs into existing C∀ behaviour by implementing necessary *sized* assertions needed by other types. (*sized* implies a concrete *vs.* abstract type with a runtime-available size, exposed as **sizeof**.) As a result, there is significant programming safety by making the size accessible and implicit, compared with C's calloc and non-array supporting memalign, which take an explicit length parameter not managed by the type system.

Figure 3.1 shows an example using function f, illustrating how dynamic values are fed into the array type. Here, the dimension of arrays x, y, and result is specified from a command-line value, dim, and these arrays are allocated on the stack. Then the x array is initialized with decreasing values, and the y array with amounts offset by constant 0.005, giving relative differences within tolerance initially and diverging for later values. The program main is run (bottom of the figure) with inputs 5 and 7 for sequence lengths. The loops follow the familiar pattern of using the variable dim to iterate through the arrays. Most importantly, the type system implicitly captures dim at the call of f and makes it available throughout f as N. The example shows dim adapting into a type-system managed length at the declarations of x, y, and result; N adapting in the same way at f's loop

```
forall( [N] ) array( bool, N ) & f( array( float, N ) &, array( float, N ) & );
int main( int argc, char * argv[] ) {
    const int dim = ato( argv[1] );             // deduce conversion type
    array( float, dim ) x, y;                   // VLAs
    for ( i; dim ) {                            // initialize arrays
        x[i] = 3.14 / (i + 1);
        y[i] = x[i] + 0.005 ;
    }
    array( bool, dim ) & result = f( x, y );    // call
    sout | "result:  " | nonl;                  // print result
    for ( i; dim )
        sout | result[i] | nonl;
    sout | nl;
    free( &result );                            // free result storage
}
$ ./a.out 5
result: true true true false false
$ ./a.out 7
result: true true true false false false false
```

Figure 3.1: Example of calling a dimension-generic function

bound; and a pass-thru use of dim at f's declaration of ret. Except for the lifetime-management issue of result, *i.e.*, explicit free, this program has eliminated both the syntactic and semantic problems associated with C arrays and their usage. The result is a significant improvement in safety and usability. In summary:

- [N] within a **forall** declares the type variable N to be number that the type system manages.
- N can be used in an expression with type size_t within the function body.
- The value of an N-expression is the acquired length, derived from the usage site, *i.e.*, generic declaration or function call.

Figure 3.2 shows N is not the same as a size_t declaration in a C++ **template**. It illustrates these differences:

1. The C++ template N can only be a compile-time value, while the C∀ N may be a runtime value.
2. C++ does not allow a template function to be nested, while C∀ allows polymorphic functions be nested. Hence, C++ precludes a simple form of information hiding.
3. The C++ template N must be passed explicitly at the call, unless N has a default value, even when C++ can deduce the type of T. The C∀ N is part of the array type and passed implicitly at the call.
4. C/C++ arrays cannot be copied, while C∀ arrays can be copied, making them a first-class object (although array copy is often avoided for efficiency).

The C++ template std::array tries to accomplish a similar mechanism to C∀ array. It is an aggregate type with the same semantics as a **struct** holding a C-style array (see Section 2.3.6, p. 23), which mitigates points 3 and 4.

```
template< typename T, size_t N >
void copy( T ret[N], T x[N] ) {
    for ( int i = 0; i < N; i += 1 ) ret[i] = x[i];
}
void demo() {
    const size_t  n = 10;
    int ret[n], x[n];                   // static size
    for ( int i = 0; i < n; i += 1 ) x[i] = i;
    copy<int, n >( ret, x );
    for ( int i = 0; i < n; i += 1 ) cout << ret[i] << ' ';
    cout << endl;
}
int main() {
    demo();
}
```

```
int main() {
    forall( T, [N] )                    // nested function
    void copy( array( T, N ) & ret, array( T, N ) & x ) {
        for ( i; N ) ret[i] = x[i];
    }
    void demo( const size_t n ) {

        array( int, n ) ret, x;         // dynamic-fixed size
        for ( i; n ) x[i] = i;          // initialize
        copy( ret,  x );                // alternative ret = x
        for ( i; n ) sout | ret[i] | nonl;
        sout | nl;
    }
    size_t  n;   sin | n;               // obtain size
    demo( n );
}
```

Figure 3.2: N-style parameters, for C++ template *vs.* C∀ generic type

Just as the first example in Section 3.1, p. 39 shows a compile-time rejection of a length mismatch, so are length mismatches stopped when they involve dimension parameters. While Figure 3.1 shows successfully calling a function f expecting two arrays of the same length,

```
array( bool, N ) & f( array( float, N ) &, array( float, N ) & );
```

a static rejection occurs when attempting to call f with arrays of differing lengths.

```
array( float, 10 ) x;
array( float, 20 ) y;
f( x, x );  // ok
f( y, y );  // ok
f( x, y );  // error
```

When the argument lengths themselves are statically unknown, the static check is conservative and, as always, C∀'s casting lets the programmer use knowledge not shared with the type system.

```
forall( [M], [N] )
void f( array(float, M) &x, array(float, N) &y ) {
    f( x, x );          // ok
    f( y, y );          // ok
    f( x, y );          // error
    if ( M == N ) f( x, (array( float, M ) &)y );  // ok
}
```

This static check's rules are presented in Section 3.4, p. 48.

Orthogonally, the C∀ array type also works inside generic *types*. That is, an array as a generic **struct**'s member and an [N] can parameterize a **struct**. The same argument safety, and the associated implicit length communication, occur. Preexisting C∀ allowed aggregate types to be generalized with type parameters, enabling parameterizing of element types. As polymorphism on functions was extended to allow parameterizing over dimensions, so is polymorphism on types. The result has profound consequences for safe memory management.

```
forall( [C], [S] )                          // num courses, num students
struct School {
    array( string, C ) course_codes;        // nested VLAs
    array( string, S ) student_ids;
    array( int, C, S ) preferences;         // matrix, C x S
};
```

Figure 3.3: C∀ Dynamic Array Member (DAM) declaration

C's Flexible Array Member (FAM) pattern [25, § 6.7.2.1.18] gives an unsafe way to put an array of dynamic length inside a structure.

```
struct S {
    ...
    double d[]; // incomplete array type ⇒ flexible array member
};
struct S * s = malloc( sizeof( struct S ) + sizeof( double [10] ) );
```

This type-S declaration defines a quasi-VLA named d as the last structure member. The variable-s initialization sizes this particular array at 10 elements. The choice of malloc here is the common-case FAM idiom, but alloca can be substituted, resulting in stack allocation. A C∀ [N]-parameterized structure improves on this pattern.

C∀ allows length-parameterized array members to be nested at arbitrary locations, with further member declarations interleaved. The result is a *Dynamic Array Member (DAM)* as illustrated in Figure 3.3. The example's School structure has course- and student-level metadata (in the respectively named fields) and a position-based preferences matrix. The structure's layout is dynamic; the starting offset of student_ids varies according to the generic parameter C, which affects the size of the course_codes field. Similarly, the offset of preferences varies by both parameters.

Figure 3.4 shows a program using the School type, along with results at different array sizes. The school variable holds many students' course-preference forms. It is on the stack and its initialization does not use any casting or size arithmetic. When heap allocation is preferred, the original pattern still applies, with the same qualities.

```
School( classes, students ) * sp = alloc();
```

The example program prints the courses in each student's preferred order, all using the looked-up display names. In the example, function getPref returns, for the student at position is, the position of their prefth-favoured class. This function helps the program to output each student's ordered preferences. The program's inputs and outputs are given on the right, for two different school sizes.

Much of this C∀ DAM ability is impossible with a C FAM. In C, they can only occur at the end of a structure; C∀ allows them anywhere. As a consequence, C∀ allows several of them, while C is limited to one. The C∀ ability to avoid casting and size arithmetic improves safety and usability over C. While C offers the *VLA* language feature, to eliminate casting and size arithmetic for stack-allocated *simple arrays*, this feature does not extend to stack-allocating a FAM-structure, for which the user must explicitly compute a total byte-size. The C∀ DAM, by contrast, treats every stack

```
forall( [C], [S] )
int getPref( School( C, S ) & school,
             int is, int pref ) {
    for ( ic; C ) {
        if ( pref == school.preferences[ic][is] )
            return ic; // offset calculation implicit
    }
    assert( false );    // must find a match
}
int main() {
    int nc, ns;
    sin | nc | "  courses,";
    sin | ns | "  students";
    School( nc, ns ) school;
    // ... elided: read data into school
    for ( is; ns ) {
        sout | school.student_ids[is] | ":   ";
        for ( pref; 1 ~= nc ) {
            int ic = getPref( school, is, pref );
            sout | school.course_codes[ ic ] | nonl;
        }
        sout | nl;
    }
}
```

```
$ cat school1
3 courses, 2 students
c\s          90111111 90222222
ENGL101             3        2
PHYS101             1        3
CHEM101             2        1
$ ./a.out < school1
90111111:
PHYS101 CHEM101 ENGL101
90222222:
CHEM101 ENGL101 PHYS101
$ cat school2
5 courses, 3 students
c\s          90111111 90222222 90333333
ENGL101             3        2        3
PHYS101             1        3        4
CHEM101             2        1        5
PHIL101             4        5        2
MATH499             5        4        1
$ ./a.out < school2
90111111:
PHYS101 CHEM101 ENGL101 PHIL101 MATH499
90222222:
CHEM101 ENGL101 PHYS101 MATH499 PHIL101
90333333:
MATH499 PHIL101 ENGL101 PHYS101 CHEM101
```

Figure 3.4: C∀ Dynamic Array Member (DAM) usage. Example program (left) with input and output (right).

allocation as a simple variable declaration. When a function operates on a C∀ DAM structure, the type system handles its memory layout transparently.

3.3 Single Dimension Array Implementation

The core of the preexisting C∀ compiler already has the "heavy equipment" needed to provide the feature set just reviewed. To apply this equipment in tracking array lengths, I encoded a dimension (array's length) as a type. The type in question does not describe any data that the program actually uses at runtime. It simply carries information through intermediate stages of C∀-to-C lowering. And this type takes a form such that, once *it* gets lowered, the result does the right thing. This new dimension type, encoding the array dimension within it, is the first limited *dependent type* [21] added to the C∀ type-system and is used at appropriate points during type resolution.

Discussion of the C∀ array implementation is broken into single- and multi-dimensional features. The multidimensional feature set of Section 3.5, p. 53 is a toolkit for managing repeated application of the single-dimensional features. Details of the array type are introduced as needed. The single-dimension discussions begins with a simple example stripped of the array type.

```
forall( [N] )
struct array_1d_float {
    float items[N];             // float elements
};
forall( T, [N] )
struct array_1d {
    T items[N];                 // any-typed elements
};
```

These two structure patterns, plus a subscript operator, is all that array provides. My main work is letting a programmer define such a structure (one whose type is parameterized by [N]) and define functions that operate on it (these being parameterized similarly).

The repurposed heavy equipment is

- The "resolver" compiler pass, which provides argument values for a declaration's type-system parameters, gathered from type information in scope at the usage site.
- The "box" compiler pass, which encodes information about type parameters into "extra" regular parameters on declarations, and corresponding arguments on calls. Notably, it conveys the size of a type foo as a __sizeof_foo parameter, and rewrites the **sizeof**(foo) expression as __sizeof_foo, *i.e.*, a use of the parameter.

The rules for resolution had to be restricted slightly, in order to achieve important refusal cases. This work is detailed in Section 3.4, p. 48. However, the resolve-box scheme, in its preexisting state, is equipped to work on (desugared) dimension parameters. The following discussion explains the desugaring and why its lowered code is correct.

A simpler structure, and a toy function on it, demonstrate what is needed for the encoding.

```
forall( [N] ) {                         // [1]
    struct thing {};
    void f( thing(N) ) { sout | N; }    // [2], [3]
}
int main() {
    thing( 10 ) x;  f( x );             // prints 10, [4]
    thing( 100 ) y;  f( y );            // prints 100
    const size_t n = ask_user();
    thing( n ) z;  f( z );              // prints user-given value, [5]
}
```

This example has:

1. The symbol N being declared as a type variable (a variable of the type system).
2. The symbol N being used to parameterize a type.
3. The symbol N being used as an expression (value).
4. A value like 10 or n being used as an argument to the parameter N.
5. The unknown-n case is written no differently than the known-n cases.

The chosen solution is to encode the value N *as a type*, so items 1 and 2 are immediately available for free. Item 3 needs a way to recover the encoded value from a (valid) type and to reject invalid types. Item 4 needs a way to produce a type that encodes the given value.

Because the box pass handles a type's size as its main datum, the encoding is chosen to use it. The production and recovery are then straightforward.

- The value n is encoded as a type whose size is n.
- Given a dimension expression e, produce an internal type **char**[e] to represent it. If e evaluates to n then the encoded type has size n.
- Given a type T (produced by these rules), recover the value that it represents with the expression **sizeof**(T). If T has size n then the recovery expression evaluates to n.

This desugaring is applied in a translation step before the resolver. The "validate" pass hosts it, along with several other canonicalizing and desugaring transformations (the pass's name notwithstanding). The running example is lowered to:

```
forall( N * ) {                                 // [1]
    struct thing { ... };
    void f( thing(N) ) { sout | sizeof(N); }    // [2], [3]
}
int main() {
    thing( char[10] ) x;  f( x );               // prints 10, [4]
    thing( char[100] ) y;  f( y );              // prints 100
    const size_t n = ask_user();
    thing( char[n] ) z;  f( z );                // prints user-given value, [5]
}
```

Observe:

1. N is now declared to be a type. It is declared to be *sized* (by the $*$), meaning that the box pass shall do its **sizeof**(N)→__sizeof_N extra parameter and expression translation.
2. thing(N) is a type; the argument to the generic thing is a type (type variable).
3. The sout... expression (being an application of the ?|? operator) has a second argument that is an ordinary expression.
4. The type of variable x is another thing(−) type; the argument to the generic thing is a type (array type of bytes, **char**[e]).
5. The dynamic still has the same translation handling as the static case, but the dynamic case's type argument is a VLA type.

From this point, preexisting C∀ compilation takes over lowering it the rest of the way to C. Here the result shows only the relevant changes of the box pass (as informed by the resolver), leaving the rest unadulterated:

```
void f( size_t __sizeof_N, void * ) { sout | __sizeof_N; } // [1], [2], [3]
int main() {
    struct __conc_thing_10 { ... } x;  f( 10, &x );  // prints 10, [4]
    struct __conc_thing_100 { ... } y;  f( 100, &y );  // prints 100
    const size_t n = ask_user();
    char __buf_z[ CALC( char[n] ) ];  f( n, __buf_z );  // prints user-given value, [5]
}
```

Observe:

1. The type parameter N is gone.
2. The type thing(N) is (replaced by **void** *, but thereby effectively) gone.
3. The sout... expression has a regular variable (parameter) usage for its second argument.
4. Information about the particular thing instantiation (value 10) is moved, from the type, to a regular function-call argument.
5. Different pre-existing translations kick in, between the dynamic and static cases, for stack-allocating the thing structure. The greater complexity of the dynamic case is not detailed here, and is place-held by the CALC fragment, which calculates the size of thing(WHAT), as a function of **sizeof**(WHAT). CALC is not affected by WHAT being dynamically *vs.* statically sized and GCC reports on **sizeof**(**char**[N]) just as well as it reports on **sizeof**(**float**). Similarly, the extra argument to the f call is obtained similarly in both cases; the dynamic case is different only in the size argument's payload being a dynamic value.

At the end of the desugaring and downstream processing, the original C idiom of "pass both a length parameter and a pointer" has been reconstructed. In the programmer-written form, only the thing is passed. The compiler's action produces the more complex form, which is error-prone to write by hand.

At the C∀ compiler's front-end parser, the changes are AST schema extensions and validation rules for enabling the sugared user input.

- Recognize the syntax **forall**(... [N] ...) as a type-variable declaration.
- Where the AST's representation of a type variable encodes its *brand*, previously including full-object types T and opaque datatypes T&, define new brand, *Dimension*. Apply this brand to the new syntax.
- Allow a type variable to occur in an expression. Validate (after parsing) that only dimension-branded type-variables are used here.
- Allow an expression to occur in type-argument position. Brand the resulting type argument as a dimension.
- Validate (after parsing), on a generic-type usage, *e.g.*, the type part of the declaration

  ```
  array_1d( foo, bar ) x;
  ```

  that the brands on the generic arguments match the brands of the declared type variables. Here, that foo is a type and bar is a dimension.

3.4 Typing of C Arrays

Essential in giving a guarantee of accurate length is the compiler's ability to reject a program that presumes to mishandle length. By contrast, most discussion so far deals with communicating length, from one party who knows it, to another willing to work with any given length. For scenarios where the concern is a mishandled length, the interaction is between two parties who both claim to know something about it.

C and C∀ can check when working with two static values.

```
enum { n = 42 };
float x[n];                                  // or just 42
float (*xp1)[42] = &x;                       // accept
float (*xp2)[999] = &x;                      // reject
warning: initialization of 'float (*)[999]' from incompatible pointer type 'float (*)[42]'
```

When a variable is involved, C and C∀ take two different approaches. Today's C compilers accept the following without a warning.

```
static const int n = 42;
float x[n];
float (* xp)[999] = &x;                      // should be static rejection here
(*xp)[500];                                  // in "bound"?
```

Here, the array x has length 42, while a pointer to it (xp) claims length 999. So, while the subscript of xp at position 500 is out of bound with its referent x, the access appears in-bound of the type information available on xp. In fact, length is being mishandled in the previous step, where the type-carried length information on x is not compatible with that of xp.

In C∀, I choose to reject this C example at the point where the type-carried length information on x is not compatible with that of xp, and correspondingly, its array counterpart at the same location:

```
static const int n = 42;
array( float, n ) x;
array( float, 999 ) * xp = &x;               // static rejection here
(*xp)[500];                                  // runtime check would pass if program compiled
```

The way the C∀ array is implemented, the type analysis for this case reduces to a case similar to the earlier C version. The C∀ compiler's compatibility analysis proceeds as:

- Is array(**float**, 999) type-compatible with array(**float**, n)?
- Is desugared array(**float**, **char**[999]) type-compatible with desugared array(**float**, **char**[n])?
- Is internal type **char**[999] type-compatible with internal type **char**[n]?

The answer is false because, in general, the value of n is unknown at compile time, and hence, an error is raised. For safety, it makes sense to reject the corresponding C case, which is a non-backwards compatible change. There are two mitigations for this incompatibility.

First, a simple recourse is available in a situation where n is *known* to be 999 by using a cast.

```
float (* xp)[999] = (float (*)[999])&x;
```

The cast means the programmer has accepted blame. Moreover, the cast is "eye candy" marking where the unchecked length knowledge is used. Therefore, a program being onboarded to C∀ requires some upgrading to satisfy the C∀ rules (and arguably become clearer), without giving up its validity to a plain C compiler. Second, the incompatibility only affects types like pointer-to-array, which are infrequently used in C. The more common C idiom for aliasing an array is to use a pointer-to-first-element type, which does not participate in C∀'s length checking.[1] Therefore, the need to upgrade legacy C code is low. Finally, if this incompatibility is a problem onboarding C

[1]Notably, the desugaring of the array type avoids letting any –[–] type decay, in order to preserve the length information that powers runtime bound-checking.

programs to C∀, it should be possible to change the C type check to a warning rather than an error, acting as a *lint* of the original code for a missing type annotation.

To handle two occurrences of the same variable, more information is needed, *e.g.*, this is fine,

```
int n = 42;
float x[n];
float (*xp)[n] = x;                    // accept
```

where n remains fixed across a contiguous declaration context. However, intervening dynamic statements can cause failures.

```
int n = 42;
float x[n];
n = 999;                               // dynamic change
float (*xp)[n] = x;                    // reject
```

As well, side-effects can even occur in a contiguous declaration context.

```
// compile unit 1
extern int n;
extern float g();
void f() {
    float x[n] = { g() };       // side−effect
    float (*xp)[n] = x;         // reject
}
```

```
// compile unit 2
int n = 42;
void g() {
    n = 999;      // accept
}
```

The issue here is that knowledge needed to make a correct decision is hidden by separate compilation. Even within a translation unit, static analysis might not be able to provide all the information. However, if the example uses **const**, the check is possible even though the value is unknown.

```
// compile unit 1
extern const int n;
extern float g();
void f() {
    float x[n] = { g() };
    float (*xp)[n] = x;     // accept
}
```

```
// compile unit 2
const int n = 42;
void g() {
    n = 999;      // reject
}
```

In summary, the new rules classify expressions into three groups:

Statically Evaluable Expressions for which a specific value can be calculated (conservatively) at compile-time. A preexisting C∀ compiler module defines which literals, enumerators, and expressions qualify and evaluates them.

Dynamic but Stable The value of a variable declared as **const**, including a **const** parameter.

Potentially Unstable The catch-all category. Notable examples include: any function-call result, **float** x[foo()], the particular function-call result that is a pointer dereference, **void** f(**const int** * n) { **float** x[*n]; }, and any use of a reference-type variable.

Within these groups, my C∀ rules are:

- Accept a Statically Evaluable pair if both expressions have the same value. Notably, this rule allows a literal to match with an enumeration value, based on the value.

- Accept a Dynamic but Stable pair if both expressions are written out the same, *e.g.*, refers to the same variable declaration.
- Otherwise, reject. Notably, reject all pairs from the Potentially Unstable group and all pairs that cross groups.

The traditional C rules are:

- Reject a Statically Evaluable pair if the expressions have two different values.
- Otherwise, accept.

Figure 3.5 gives a case-by-case comparison of the consequences of these rule sets. It demonstrates that the C∀ false alarms occur in the same cases as C treats unsafely. It also shows that C-incompatibilities only occur in cases that C treats unsafely.

These rules being conservative is the reason that Section 3.3, p. 45 calls the C∀ array type only *limited* dependent. In programming languages featuring general dependent typing, considerably more effort is invested in being able to conclude that two type-parameterizing expressions always compute the same value. While conservatism is inescapable, these languages use good approximations of eventual runtime behaviour to enable conclusions about more dynamic behaviours, such as a program never popping from an empty stack.

The conservatism of the new rule set can leave a programmer requiring a recourse, when needing to use a dimension expression whose stability argument is more subtle than current-state analysis. This recourse is to declare an explicit constant for the dimension value. Consider these two dimension expressions, rejected by the blunt current-state rules:

```
void f( int & nr, const int nv ) {
    float x[nr];
    float (*xp)[nr] = &x;          // reject: nr varying (no references)
    float y[nv + 1];
    float (*yp)[nv + 1] = &y;      // reject: ?+? unpredictable (no functions)
}
```

Yet, both dimension expressions are reused safely. The nr reference is never written, no implicit code (load) between declarations, and control does not leave the function between the uses. As well, the built-in ?+? function is predictable.

To make these cases work, the programmer adds these constant declarations:

```
void f( int & nr, const int nv ) {
    const int nx = nr;
    float x[nx];
    float (*xp)[nx] = & x;         // accept
    const int ny = nv + 1;
    float y[ny];
    float (*yp)[ny] = & y;         // accept
}
```

The result is the originally intended semantics, achieved by adding a superfluous “snapshot it as of now” directive. Note that casting does not help with this issue, because it does not express information about values at different points in time.

Case	Values Equal C	Values Equal C∀	Values Unequal C	Values Unequal C∀	Compat.
Both Statically Evaluable, Same Symbol	Accept	Accept			✓
Both Statically Evaluable, Different Symbols	Accept	Accept	Reject	Reject	✓
Both Dynamic but Stable, Same Symbol	Accept	Accept			✓
Both Dynamic but Stable, Different Symbols	Accept	Reject *	Accept !	Reject	✗
Both Potentially Unstable, Same Symbol	Accept	Reject *	Accept !	Reject	✗
Any other grouping, Different Symbol	Accept	Reject *	Accept !	Reject	✗

Legend

- Each row gives the treatment of a test harness of the form

  ```
  float x[ expr1 ];
  float (*xp)[ expr2 ] = &x;
  ```

  where expr1 and expr2 are meta-variables varying according to the row's Case. Each row's claim applies to other harnesses too, including,
  - calling a function with a parameter like x and an argument of the xp type,
  - assignment in place of initialization,
  - using references in place of pointers, and
  - for the C∀ array, calling a polymorphic function on two T-typed parameters with &x- and xp-typed arguments.
- Each case's claim is symmetric (swapping expr1 with expr2 has no effect), even though most test harnesses are asymmetric.
- The table treats symbolic identity (Same/Different on rows) apart from value equality (Equal/Unequal on columns).
  - The expressions 1, 0+1 and n (where n is a variable with value 1), are all different symbols with the value 1.
  - The column distinction expresses ground truth about whether an omniscient analysis should accept or reject.
  - The row distinction expresses the simple static factors used by today's analyses.
- Accordingly, every Reject under Values Equal is a false alarm (*), while every Accept under Values Unequal is an allowed misuse (!).

Figure 3.5: Case comparison for array type compatibility, given pairs of dimension expressions.

The snapshot trick is also used by the C∀ translation, though to achieve a different outcome. Rather obviously, every array must be subscriptable, even a bizarre one:

```
array( float, rand(10) ) x;          // x[0] => no elements
x[0];                                // 10% chance of bound-check failure
```

Less obvious is that the mechanism of subscripting is a function call, which must communicate length accurately. The bound-check above (callee logic) must use the actual allocated length of x, without mistakenly reevaluating the dimension expression, rand(10). Adjusting the example to make the function's use of length more explicit:

```
forall( T * )
void f( T * x ) { sout | sizeof( *x ); }
float x[ rand(10) ];
f( x );
```

Considering that the partly translated function declaration is, loosely,

```
void f( size_t __sizeof_T, void * x ) { sout | __sizeof_T; }
```

the translation calls the dimension argument twice:

```
float x[ rand(10) ];
f( rand(10), &x );
```

The correct form is:

```
size_t __dim_x = rand(10);
float x[ __dim_x ];
f( __dim_x, &x );
```

Dimension hoisting already existed in the C∀ compiler. However, it was buggy, particularly with determining, "Can hoisting the expression be skipped here?" for skipping this hoisting is clearly desirable in some cases—for example, when a programmer has already done manual hoisting. In the new implementation, these cases are correct, harmonized with the accept/reject criteria.

3.5 Multidimensional Array Implementation

A multidimensional array implementation has three relevant levels of abstraction, from highest to lowest, where the array occupies *contiguous memory*.

1. Flexible-stride memory: this model has complete independence between subscript ordering and memory layout, offering the ability to slice by (provide an index for) any dimension, *e.g.*, slice a row, column, or plane, *e.g.*, c[3][4][*], c[3][*][5], c[3][*][*].
2. Fixed-stride memory: this model binds the first subscript and the first memory layout dimension, offering the ability to slice by (provide an index for) only the coarsest dimension, m[row][*] or c[plane][*][*], *e.g.*, slice only by row (2D) or plane (3D).
3. Explicit-displacement memory: this model has no awareness of dimensions, just the ability to access memory at a distance from a reference point (base-displacement addressing), *e.g.*, x + 23 or x[23] $\Rightarrow$ 23rd element after the start of x. A programmer must manually build any notion of dimensions using other tools; hence, this style is not offering multidimensional arrays (see Figure 2.1, p. 18 right example).

There is some debate as to whether the abstraction point ordering above goes $\{1, 2\} < 3$, rather than my numeric ordering. That is, styles 1 and 2 are at the same abstraction level, with 3 offering a limited set of functionality. I chose to build the C∀ style-1 array upon a style-2 abstraction. Justification of the decision follows, after the description of the design.

Style 3 is an array implementation's natural target. The hardware offers this model to the C compiler, with bytes as the unit of displacement. C offers this model to programmers as pointer arithmetic, with arbitrary sizes as the unit. Casting a multidimensional array as a single-dimensional array/pointer, then using x[i] syntax to access its elements, is still a form of pointer arithmetic.

To step into the implementation of C∀'s new type-1 multidimensional arrays in terms of C's existing type-2 multidimensional arrays, it helps to clarify that the interface is low-level, *i.e.*, a C/C∀ array interface includes memory-layout detail. Specifically, the defining requirement of a type-1 system is the ability to slice a column from a column-finest matrix. Hence, the required memory shape of such a slice is fixed, before any discussion of implementation. The implementation presented here is how the C∀ array-library wrangles the C type system to make it do memory steps that are consistent with this layout while not affecting legacy C programs.

The new C∀ standard-library array-datatype supports richer multidimensional features than C. The C∀ array implementation follows C's contiguous approach, *i.e.*, **float** [r][c], with one contiguous object subscripted by coarsely-strided dimensions directly wrapping finely-strided dimensions. Beyond what C's array type offers, the C∀ array brings direct support for working with a *noncontiguous* array slice, allowing a program to work with dimension subscripts given in a non-physical order.

The following examples use the matrix declaration array(**float**, 5, 7) m, loaded with values incremented by 0.1, when stepping across the length-7 finely-strided column dimension, and incremented by 1, when stepping across the length-5 coarsely-strided row dimension.

r\c	0	1	2	3	4	5	6
0	0.0	0.1	0.2	0.3	0.4	0.5	0.6
1	1.0	1.1	1.2	1.3	1.4	1.5	1.6
2	2.0	2.1	2.2	2.3	2.4	2.5	2.6
3	3.0	3.1	3.2	3.3	3.4	3.5	3.6
4	4.0	4.1	4.2	4.3	4.4	4.5	4.6

The memory layout is 35 contiguous elements with strictly increasing addresses.

A trivial form of slicing extracts a contiguous inner array, within an array-of-arrays. As for the C array, a lesser-dimensional array reference can be bound to the result of subscripting a greater-dimensional array by a prefix of its dimensions, *e.g.*, m[2], giving the third row. This action first subscripts away the most coarsely strided dimensions, leaving a result that expects to be subscripted by the more finely strided dimensions, *e.g.*, m[2][3], giving the value 2.3. The following is an example slicing a row.

```
forall( [N] )
void print1d_cstyle( array( float, N ) & r ) {   // C style
    for ( i; N ) sout | r[i] | nonl;
    sout | nl;
}
print1d_cstyle( m[ 2 ] );                        // row 2: 2.0 2.1 2.2 2.3 2.4 2.5 2.6
```

However, function print1d_cstyle is asserting too much knowledge about its parameter r for printing either a row slice or a column slice. Specifically, declaring the parameter r with type array means that r is contiguous, which is unnecessarily restrictive. That is, r need only be of a collection type that offers a subscript operator (of type ptrdiff_t $\rightarrow$ **float**) with managed length N. The C∀ array library provides the trait ar, defined as follows. With it, the original declaration can be generalized with the same body.

```
// from array.hfa, simplified:
forall( A &,                                // array-like type
    Tv &,                                   // element (type) of A
    [N] )                                   // length (value) of A
trait ar {
    Tv & ?[?]( A &, size_t );               // myA[42]: Tv
    size_t len( A & );                      // len(myA): size_t
    REPRESENTATION_ELIDED( A, N );          // records relationship: managed len of A is N
};
// user's program, revised:
forall( [N], C & | ar( C, float, N ) )      // add trait
void print1d( C & c );
print1d( m[ 2 ] );                          // row: 2.0 2.1 2.2 2.3 2.4 2.5 2.6
```

The nontrivial slicing in this example now allows passing a *noncontiguous* slice to print1d, where the C∀ array library provides a "subscript by all" operation for this purpose. In a multi-dimensional subscript operation, any dimension given as all is a placeholder, *i.e.*, "not yet subscripted by a value", waiting for a value implementing the ar trait.

```
print1d( m[ 2, all ] );                     // row 2: 2.0 2.1 2.2 2.3 2.4 2.5 2.6
print1d( m[ all, 3 ] );                     // column 3: 0.3 1.3 2.3 3.3 4.3
```

The example shows x[2] and x[2, all] both refer to the same, "2.*" slice. Indeed, the various print1d calls under discussion access the entry with value 2.3 as x[2][3], x[2,all][3], and x[all,3][2]. This design preserves (and extends) C array semantics by defining x[i,j] to be x[i][j] for numeric subscripts, but also for "subscripting by all". That is:

x[2,all][3]	≡	x[2][all][3]	≡	x[2][3]	(here, all is redundant)
x[all,3][2]	≡	x[all][3][2]	≡	x[2][3]	(here, all is effective)

Narrating progress through each of the −[−][−][−][2] expressions gives, firstly, a definition of −[all], and secondly, a generalization of C's −[i]. Where all is redundant:

x	2-dimensional, want subscripts for coarse then fine
x[2]	1-dimensional, want subscript for fine; lock coarse == 2
x[2][all]	1-dimensional, want subscript for fine
x[2][all][3]	0-dimensional; lock fine == 3

Where all is effective:

x	2-dimensional, want subscripts for coarse then fine
x[all]	2-dimensional, want subscripts for fine then coarse
x[all][3]	1-dimensional, want subscript for coarse; lock fine == 3
x[all][3][2]	0-dimensional; lock coarse == 2

The semantics of −[all] is to dequeue from the front of the "want subscripts" list and re-enqueue at its back. For example, in a two dimensional matrix, this semantics conceptually transposes the matrix by reversing the subscripts. The semantics of −[i] is to dequeue from the front of the "want subscripts" list and lock its value to be i.

Contiguous arrays, and slices of them, are all represented by the same underlying parameterized

[2]The first "−" is a variable expression and the remaining "−" are subscript expressions.

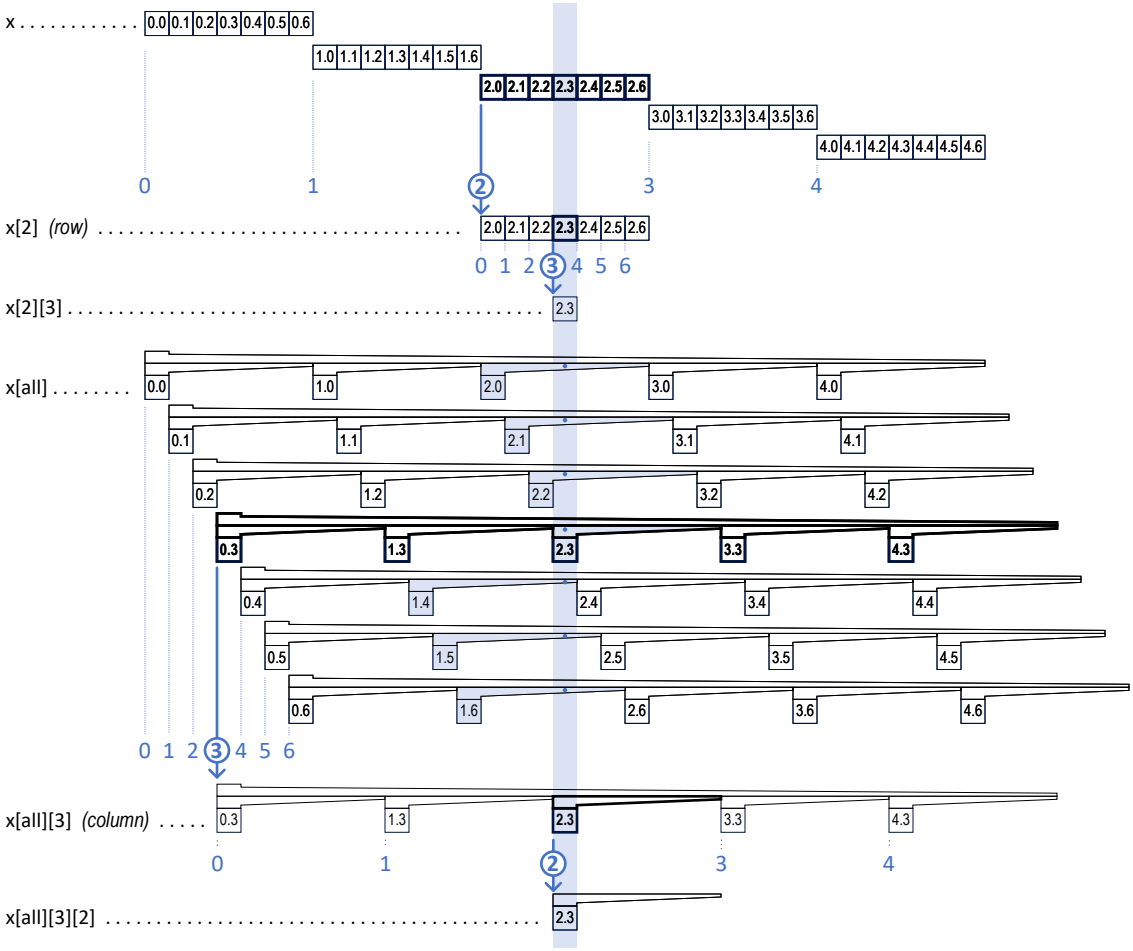


Figure 3.6: Visualization of subscripting by value and by 'all.' Uses example variable x of type array(**float**, 5, 7), understood as 5 rows by 7 columns. The horizontal layout represents contiguous memory addresses while the vertical layout is conceptual. The vertical shaded band highlights the location of the targeted element, 2.3. Any such vertical slice contains various interpretations of a single address.

type, which includes stride information in its metadata. The –[all] operation takes subscripts, *e.g.*, x[2][all], x[all][3], *etc.*, and converts (transposes) from the base reference x[all] to a specific reference of the appropriate form. The running example's all-effective step, stated more concretely, is:

x : 5 of (7 of **float** each spaced 1 **float** apart) each spaced 7 floats apart
x[all] : 7 of (5 of **float** each spaced 7 **float**s apart) each spaced 1 **float** apart

Figure 3.6 shows one element (in the shaded band) accessed two different ways: as x[2][3] and as x[all][3][2]. In both cases, subscript 2 selects from the coarser dimension (rows of x), while subscript 3 selects from the finer dimension (columns of x). The figure illustrates the value of each subexpression, comparing how numeric subscripting proceeds from x, *vs.* from x[all]. Proceeding from x gives the numeric indices as coarse then fine, while proceeding from x[all] gives them fine then coarse. These two starting expressions, which are the example's only multidimensional subexpressions (those that received zero numeric indices so far), are illustrated with vertical steps where a *first* numeric index selects.

The figure's presentation offers an intuitive answer to: What is an atomic element of x[all]?

1. x[all] itself is simply a two-dimensional array, in the strict C sense, of these building blocks.
2. An array of these atoms (like the intermediate x[all][3]) is just a contiguous arrangement of them, done according to their size; call such an array a column. A column is almost ready to be arranged into a matrix; it is the *contained value* of the next-level building block, but another lie about size is required.
3. An atom (like the bottommost value, x[all][3][2]), is the contained value (in the square box) and a lie about its size (the left diagonal above it, growing upward).

At first, an atom needs to be arranged as if it is bigger, but now a column needs to be arranged as if it is smaller (the left diagonal above it, shrinking upward). These lying columns, arranged contiguously according to their size (as announced) form the matrix x[all]. Because x[all] takes indices, first for the fine stride, then for the coarse stride, it achieves the requirement of representing the transposition of x. Yet every time the programmer presents an index, a C-array subscript is achieving the offset calculation.

In the x[all] case, after the finely strided subscript is done (column 3 is selected), the locations referenced by the coarse subscript options (rows 0..4) are offset by 3 floats, compared with where analogous rows appear when the row-level option is presented for x. For example, in x[all], the shaded band and its immediate values to the left touches atoms 2.3, 2.0, 2.1, 2.2, 1.4, 1.5 and 1.6. But only the atom 2.3 is storing its value there. The rest are lying about (conflicting) claims on this location, but never exercising these alleged claims.

Lying is implemented as casting. The arrangement just described is implemented in the structure arpk. This structure uses one type in its internal field declaration and offers a different type as the return of its subscript operator. The field within is a plain-C array of the fictional type, which is 7 floats long for x[all][3][2] and 1 float long for x[all][3]. The subscript operator presents what is really inside, by casting to the type below the left diagonal of the lie.

Casting, overlapping, and lying are unsafe. The mission is to implement a style-1 feature in the type system for safe use by a programmer. The offered style-1 system is allowed to use unsafe internally, just as C's implementation of a style-2 system (upon a style-3 system) does, even when the programmer is using it without casts or pointer arithmetic. Having a style-1 system relieves the programmer from resorting to unsafe pointer arithmetic when working with noncontiguous slices.

The C∀ array library defines types and operations that ensure proper elements are accessed soundly in spite of the overlapping. The arpk structure and its –[i] operator are defined in Figure 3.7. The private arpk structure (array with explicit packing) is generic over four parameters: dimension length value, masquerading-as type, immediate-element type and base-element type. This structure's public interface is hidden behind the array(...) macro and the subscript operator. Construction by array initializes the masquerading-as type information to be equal to the contained-element information. Subscripting by all rearranges the order of masquerading-as types to achieve, in general, nontrivial striding. Subscripting by a number consumes the masquerading-as size of the contained element type, does normal array stepping according to that size, and returns the element found there, in unmasked form.

An instantiation of the arpk generic is given by the array(E_base, N0, N1, ...) expansion, which is arpk(N0, Rec, Rec, E_base), where Rec is array(E_base, N1, ...). In the base case, array(E_base)

```
forall(
    [N],                                    // length of current dimension
    S & | sized(S),                         // masquerading-as
    Timmed &,                               // immediate element, often another array
    Tbase &                                 // base element, e.g., float, never array
) { // distribute forall to each element
    struct arpk {
        S strides[N];                       // so that sizeof(this) is N of S
    };
    // expose Timmed, stride by S
    static inline Timmed & ?[?]( arpk( N, S, Timmed, Tbase ) & a, long int i ) {
        subcheck( a, i, 0, N );
        return (Timmed &)a.strides[i];
    }
}
```

Figure 3.7: Definition of structure 'arpk,' which underlies the C∀ array

is just E_base. Because this construction uses the same value for the generic parameters S and E_im, the resulting layout has trivial strides.

Subscripting by all, to operate on nontrivial strides, is a dequeue-enqueue operation on the E_im chain, which carries S instantiations, intact, to new positions. Expressed as an operation on types, this rotation is:

$$\begin{aligned} suball(arpk(N, S, E_i, E_b)) &= enq(N, S, E_i, E_b) \\ enq(N, S, E_b, E_b) &= arpk(N, S, E_b, E_b) \\ enq(N, S, arpk(N', S', E'_i, E_b), E_b) &= arpk(N', S', enq(N, S, E'_i, E_b), E_b) \end{aligned}$$

The use of the word "all" has roots in the similarity of C∀'s x[all, j, all] to the canonical linear-algebraic notation $\mathbf{X}_{:j:}$ of [34, Fig. 2.2]. However, the association of x[I, J, K] with x[I][J][K] is this work's desire to preserve a C idiom. By C syntactic rules, this expression could also be written (x[I])[J][K]. Therefore, x[all, j, all] is (x[all])[j][all]; so, x[all] needs a meaning. The generalized transpose presented earlier is the solved-for meaning of x[all] that gives y$\stackrel{\text{def}}{=}$x[all, j, all] the meaning whereby y[i,k] is x[i,j,k], as desired.

3.6 Zero Overhead

At runtime, the C∀ array is exactly a C array. C∀ array subscripting is protected with runtime bound checks. The array dependent-typing provides information to the C optimizer, allowing it to remove many of the bound checks. This section provides a demonstration of these effects.

This section therefore offers an assurance that the C∀ array performs well. In the linked-list and string chapters following, the performance claim is fundamentally different from the present one (and somewhat more traditional). The C∀ linked list and string come from libraries of runtime

code; their libraries need to be compared with other libraries. So, those chapters develop, measure and analyze microbenchmarks to establish performance. Here, the claim is not that a C∀ code path has acceptable quantitative attributes, relative to a qualitatively different code path. Rather, the claim for arrays is that the value-adding mechanisms are erased before runtime (in appropriate cases). This claim is (a) stronger and (b) logical, rather than quantitative. Therefore, the array case is made by code inspection, whereas the linked-list and string cases are made empirically. The future-work discussions (for arrays, 3.9.5, p. 77 and 3.9.6, p. 77) address limitations of the presented assurances.

The experiment compares the C∀ array system with the simple-safety system most typically exemplified by Java arrays (Section 3.8.2, p. 69), and reflected in C++ vector using member .at for subscripting (Section 3.8.3, p. 72). The essential feature is the one-to-one correspondence between array instances and the symbolic bounds on which dynamic checks are based. Hence, it is the implicit C∀ checked subscripting and explicit C++ vector .at mechanisms being explained and not limitations of C++ optimization.

Observing whether a runtime bound check is present or absent occurs by reading compiler-generated assembly code. This measure is necessary because the GCC back-end removes a redundant bound check, in all cases. In the C++ case, the bound check is injected by its code occurring in the C++ library's implementation of vector::at; for C∀, it comes similarly from the C∀ array's subscript operator. Though C++ does not offer such a tempting intermediary, C∀ offers its lowered C form, less succeptible to the particulars of a fickle back-end. Unfortunately, C∀'s lowered C still has bound checks on *every* subscript. In fact, the C lowering has every subscript appearing as a call to a (inline-marked) function, whose (sole) body runs a bound check. Only after inlining has occurred (by back-end optimizer) can condition constant-folding driven by value-range analysis determine that a complain-out-of-bound case is unreachable. The latter techniques are available a priori in the back-end optimizer, while not so in the C∀ source-to-source translator; indeed, arguably, they are inappropriate responsibilities for it. So, the removability of a given bound check is determined by observing optimized generated assembly code.

This approach does limit the generalizability of the results. In mitigation, the GCC-x86[3] experiment, whose presentation follows, was repeated also on ARM[4] and with Clang[5][6]. Similar results were observed on both, though these assembly-code analyses are elided from this presentation. Because the results generalize across a few common toolchains, the experiment sets a reasonable expectation about how the respective array implementations feed a *typical* optimizer.

As a control case, a simple loop receives the same optimization treatment in both the C∀ and C++ versions. When the surrounding code makes it clear subscripting is always in bound, no

[3]See notes to Figure 3.8, p. 61 for specifics.

[4]GCC 13.3.0 targeting aarch64-linux-gnu with performance-relevant flags -O3; the C++ vector implementation is from libstdc++ of GCC 13.3.0.

[5]Clang 19.1.1 targeting x86_64-linux-gnu with performance-relevant flags -O3; the C++ vector implementation is from libc++ of LLVM 19.1.1.

[6]C∀ does not actually support Clang as a C back-end; a program's lowered form uses GCC extensions not available in Clang. But this loop-subscript demo does not use the C∀ features that need the GCC extensions. A version of the lowering that Clang accepts was obtained by manually editing the C∀ source (replacing C∀ sout printing with printf), generating a prelude-free lowered C form, restoring the few necessary missing prelude declarations, removing demo-unused more-complex C∀ array definitions, and simplifying how array construction/destruction occurs (which is outside of the loops being analyzed, *i.e.*, off the demo's hot path).

dynamic bound check is observed in the program's optimized assembly code. But when the environment is adjusted, such that the subscript is possibly invalid, the bound check appears in the optimized assembly, ready to catch a mistake.

Figure 3.8 gives a control-case example of summing values in an array, where each column shows the program in languages C (a, d, g), C∀ (b, e, h), and C++ (c, f, i). The C code has no bounds checking, while the C∀ and C++ can have bounds checking. The source-code functions in (a, b, c) can be compiled to have either correct or incorrect uses of bounds. When compiling for correct bound use, the BND macro passes its argument through, so the loops cover exactly the passed array sizes. When compiling for possible incorrect bound use (a programming error), the BND macro hardcodes the loops for 100 iterations, which implies out-of-bound access attempts when the passed array has fewer than 100 elements. The assembly code excerpts show optimized translations for correct-bound mode in (d, e, f) and incorrect-bound mode in (g, h, i). Because incorrect-bound mode hardcodes 100 iterations, the loop always executes a first time, so this mode does not have the n == 0 branch seen in correct-bound mode. For C, this difference is the only (d–g) difference. For correct bounds, the C∀ translation equals the C translation, *i.e.*, there is no (d–e) difference, while C++ has an additional indirection to dereference the vector's auxiliary allocation.

Differences arise when the bound-incorrect programs are passed an array shorter than 100. The C version, (g), proceeds with undefined behaviour, reading past the end of the array. The C∀ and C++ versions, (h, i), flag the mistake with the runtime check, the i >= n branch. This check is provided implicitly by the library types array and vector, respectively. The significant result is the *absence* of runtime checks from the bound-correct translations, (e, f). The code optimizer has removed them because, at the point where they would occur (immediately past .L28/.L3), it knows from the surrounding control flow either i == 0 && 0 < n or i < n (directly), *i.e.*, i < n either way. So any repeated checks, *i.e.*, the branch and its consequent code (raise error), are removable.

When dimension sizes get reused, C∀ has an advantage over C++-vector at getting simply written programs well optimized. Figures 3.9, p. 62 shows a simple matrix multiplication over a row-major encoding. Simple means coding directly to the intuition of the mathematical definition without trying to optimize for memory layout. In the assembly code of Figure 3.10, p. 63, the looping pattern of Figure 3.8 (d, e, f), "Skip ahead on zero; loop back for next," occurs with three nesting levels. Simultaneously, the dynamic bound-check pattern of Figure 3.8 (h, i), "Get out on invalid," occurs, targeting .L7, .L9 and .L8 (bottom right). Here, Figures 3.10, p. 63 shows the C∀ solution optimizing into practically the C solution, while the C++ solution shows added runtime bound checks. Like in Figure 3.8 (e), the significance is the *absence* of library-imposed runtime checks, even though the source code is working through the the C∀ array library. The optimizer removed the library-imposed checks because the data structure array-of-array is constrained by its type to be shaped correctly for the intuitive looping. In C++, the same constraint does not apply to vector-of-vector. Because every individual vector carries its own size, two types of bound mistakes are possible.

In detail, the three structures received may not be matrices at all, per the obvious/dense/total interpretation; rather, any one might be ragged-right in its rows. The C∀ solution guards against this possibility by encoding the minor length (number of columns) in the major element (row) type. In res, for example, each of the M rows is array(**float**, N), guaranteeing N cells within it. Or more technically, guaranteeing N as the basis for the imposed bound check *of every row.* As well, the C++

C

```
double sum(
      size_t n, float x[] ) {
   double sum = 0;
   for ( size_t i = 0;
         i < BND( n ); i++ )
      sum += x[i];
   return sum;
}
```

(a)

C∀

```
forall( [N] )
double sum(
      array(float, N) & x ) {
   double sum = 0;
   for ( i;
         BND( N ) )
      sum += x[i];
   return sum;
}
```

(b)

C++ (nested vector)

```
double sum(
      vector<float> & x ) {
   double sum = 0;
   for ( size_t i = 0;
         i < BND( x.size() ); i++ )
      sum += x.at(i);
   return sum;
}
```

(c)

```
      testq   %rdi, %rdi
      je      .L4
      leaq    (%rsi,%rdi,4), %rax
      pxor    %xmm0, %xmm0
.L3:
      pxor    %xmm1, %xmm1
      addq    $4, %rsi
      cvtss2sd -4(%rsi), %xmm1
      addsd   %xmm1, %xmm0
      cmpq    %rsi, %rax
      jne     .L3
      ret
.L4:
      pxor    %xmm0, %xmm0
      ret
```

(d)

```
      testq   %rdi, %rdi
      je      .L29
      leaq    (%rdx,%rdi,4), %rax
      pxor    %xmm0, %xmm0
.L28:
      pxor    %xmm1, %xmm1
      addq    $4, %rdx
      cvtss2sd -4(%rdx), %xmm1
      addsd   %xmm1, %xmm0
      cmpq    %rdx, %rax
      jne     .L28
      ret
.L29:
      pxor    %xmm0, %xmm0
      ret
```

(e)

```
      cmpq    %rdx, %rax
      je      .L4
      xorl    %eax, %eax
      pxor    %xmm0, %xmm0
.L3:
      pxor    %xmm1, %xmm1
      cvtss2sd (%rdx,%rax,4), %xmm1
      addq    $1, %rax
      addsd   %xmm1, %xmm0
      cmpq    %rcx, %rax
      jb      .L3
      ret
.L4:
      pxor    %xmm0, %xmm0
      ret
```

n == 0

i < n

(f)

```
.L2:
      pxor    %xmm1, %xmm1
      addq    $4, %rsi
      cvtss2sd -4(%rsi), %xmm1
      addsd   %xmm1, %xmm0
      cmpq    %rsi, %rax
      jne     .L2
      ret
```

(g)

```
.L28:
      cmpq    %rdi, %r8
      je      .L33
      pxor    %xmm1, %xmm1
      cvtss2sd (%rdx,%r8,4), %xmm1
      addq    $1, %r8
      addsd   %xmm1, %xmm0
      cmpq    $100, %r8
      jne     .L28
      ret
.L33:
      movq    %rdx, %rcx
      subq    $8, %rsp
      movslq  %r8d, %rsi
      movq    %r8, %rdx
      leaq    .LC9(%rip), %rdi
      xorl    %eax, %eax
      call    ...abort...
```

(h)

```
.L3:
      cmpq    %rax, %rsi
      je      .L9
      pxor    %xmm1, %xmm1
      cvtss2sd (%rdx,%rax,4), %xmm1
      addq    $1, %rax
      addsd   %xmm1, %xmm0
      cmpq    $100, %rax
      jne     .L3
      ret
.L9:
      subq    $8, %rsp
      movq    %rsi, %rdx
      leaq    .LC1(%rip), %rdi
      xorl    %eax, %eax
      call    ...throw out of range...
```

i >= n

i < 100

(i)

Figure 3.8: Bound-check overhead comparison, control case. All three programs/compilers[7] generate equally performant code when the programmer "self-checks" bounds via a loop's condition. Yet both C∀ and C++ versions generate code that is slower and safer than C's when the programmer might overrun the bounds.

C

```
void mul( size_t m,
      size_t n, size_t p,
      float res[m][n],
      float lhs[m][p],
      float rhs[p][n] ) {
   for ( size_t i = 0;
         i < m; i++ )
      for ( size_t j = 0;
            j < n; j++ ) {
         res[i][j] = 0.0;
         for ( size_t k = 0;
               k < p; k++ )
            res[i][j] +=
               lhs[i][k] *
               rhs[k][j];
      }
}
```

C∀

```
forall( [M], [N], [P] )
void mul(
      array(float, M, N) & res,
      array(float, M, P) & lhs,
      array(float, P, N) & rhs ) {
   for ( i; M )
      for ( j; N ) {
         res[i, j] = 0.0;
         for ( k; P )
            res[i, j] +=
               lhs[i, k] *
               rhs[k, j];
      }
}
```

C++ (nested vector)

```
void mul(
      vector<vector<float>> & res,
      vector<vector<float>> & lhs,
      vector<vector<float>> & rhs ) {
   for ( size_t i = 0;
         i < res.size(); i++ )
      for ( size_t j = 0;
            j < res.at(i).size(); j++ ) {
         res.at(i).at(j) = 0.0;
         for ( size_t k = 0;
               k < rhs.size(); k++ )
            res.at(i).at(j) +=
               lhs.at(i).at(k) *
               rhs.at(k).at(j);
      }
}
```

Figure 3.9: Overhead comparison, differentiation case, source.

type does not impose the mathematically familiar constraint of $(M \times P)(P \times N) \rightarrow (M \times N)$. Even assuming away the first issue, *e.g.*, that in lhs all minor/cell counts agree, the data format allows the rhs major/row count to disagree. Or, the possibility that the row count res.size() disagrees with the row count lhs.size() illustrates this bound-mistake type in isolation. The C∀ solution guards against this possibility by keeping length information separate from the array data, and therefore eligible for sharing. This capability lets C∀ escape the one-to-one correspondence between array instances and symbolic bounds, where this correspondence leaves a C++-vector programmer stuck with a matrix representation that repeats itself.

In the alternate-environment trials, GCC-ARM compilation gave identical control-flow structures as the x86 presentation just shown. Clang compilation (LLVM optimization) treated the differentiation case's three nested loops differently than GCC, though uniformly across C, C++ and C∀. The essential feature of a conditional call to a "complain about bad index" function occurs in the Clang equivalents Figures 3.8(h, i) and Section 3.10(c) but not Section 3.10(b), exactly as in the GCC analysis.

It is important to clarify that the C∀ solution does not become unsafe (like C) in losing its dynamic checks, even though it becomes fast (as C) in losing them. The dynamic checks are dismissed as unnecessary *because* the program is safe to begin with. To achieve the same performance, a C++ programmer must state appropriate assertions or assumptions, to allow the optimizer to dismiss the runtime checks. This effort would be inherently tied to choices like compiler and C++ library. The solution requires doing the work of the inner-loop checks as a *preflight step*. But this step requires looping and doing it upfront gives too much separation for the optimizer to see "has been checked already" in the deep loop. So, the programmer must restate the preflight observation within the deep loop, but this time as an unchecked assumption. Such assumptions are

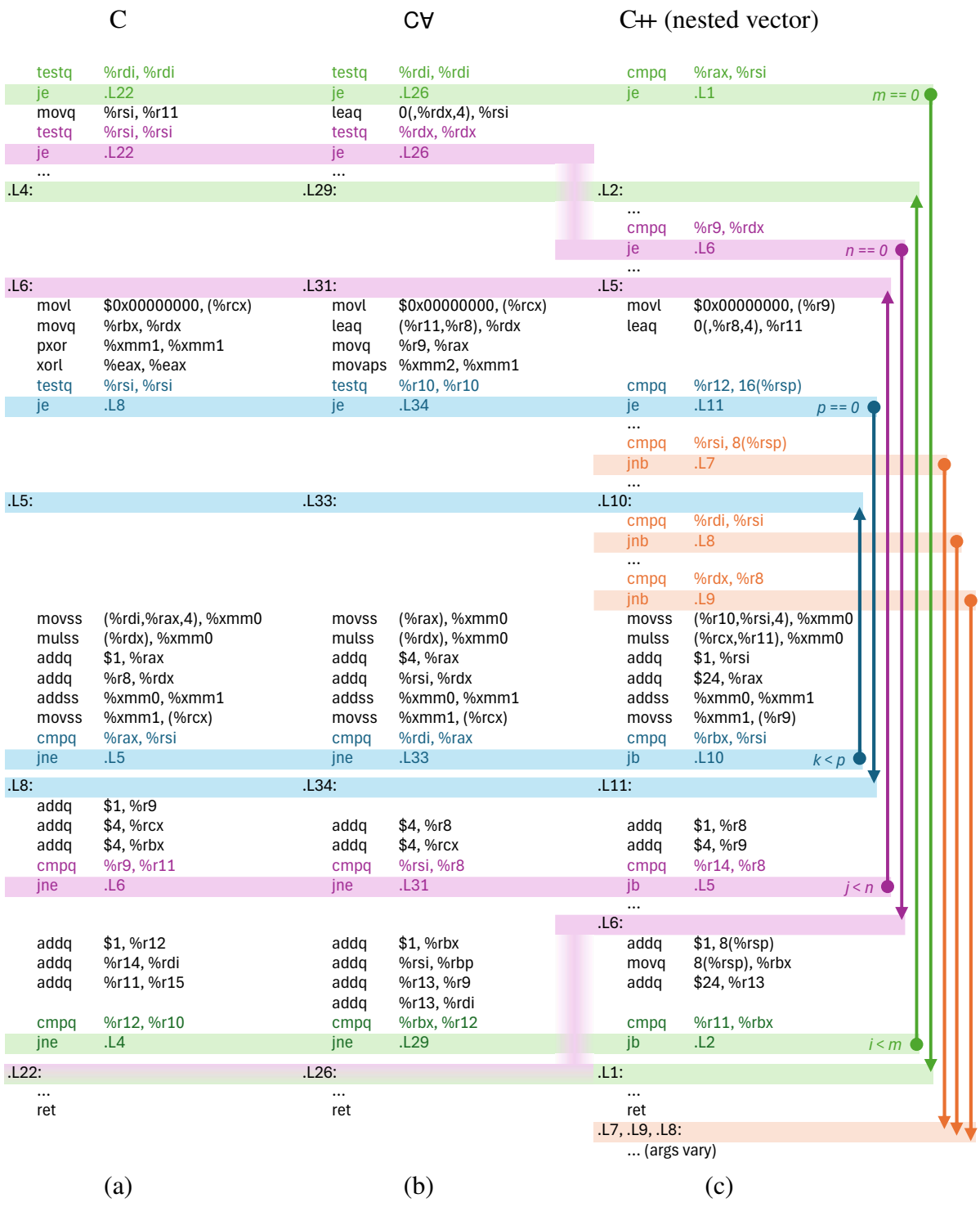


Figure 3.10: Overhead comparison, differentiation case, assembly.

risky because they introduce further undefined behaviour when misused. Only the programmer's discipline remains to ensure this work is done without error.

In summary, the C∀ solution lets a simply stated program have dynamic guards that catch bugs, while letting a simply stated bug-free program run as fast as the unguarded C equivalent.

3.7 Array Lifecycle

An array, like a structure, wraps smaller objects. Any object type, like string, can be an array element or structure member. A consequence is that the lifetime of the container must match with its contained objects: all elements and members must be initialized/uninitialized implicitly as part of the container's allocation/deallocation. Modern programming languages implicitly perform these operations via a type's constructor and destructor. Therefore, C∀ must assure that an array's elements' lifetime operations are called. Preexisting C∀ mechanisms achieve this requirement, but with poor performance. Furthermore, advanced array users need an exception to the basic mechanism, which does not occur with other aggregates. Hence, arrays introduce subtleties in supporting an element's lifecycle.

The preexisting C∀ support for contained-element lifecycle is based on recursive occurrences of the object-type (otype) pseudo-trait. A type is an otype, if it provides a default (parameterless) constructor, copy constructor, assignment operator, and destructor (like C++). For a structure with otype members, the compiler implicitly generates implementations of the four otype functions for the outer structure. Then the generated default constructor for the outer structure calls the default constructor for each member, and the other otype functions work similarly. For a member that is a C array, these calls occur in a loop for each array element, which even works for VLAs. This logic works the same whether the member is a concrete type (that happens to be an otype) or the member is a polymorphic type asserted to be an otype (which is implicit in the syntax, **forall**(T)). The C∀ array has the simplified form (similar to one seen before):

```
forall( T * )   // non-otype element, no lifecycle functions
// forall( T )  // otype element, lifecycle functions asserted
struct array5 {
    T __items[ 5 ];
};
```

Being a structure with a C-array member, the otype-form declaration T causes array5(**float**) to be an otype too. But this otype-recursion pattern has a performance issue. Figure 3.11 shows the steps to find lifecycle functions, under the otype-recursion pattern, for a cube of **float**:

```
array5( array5( array5( float ) ) )
```

The work needed for the full **float**-cube is 256 leaves. Then the otype-recursion pattern generates helper functions and thunks that are exponential in the number of dimensions. From subjective experience, compile times are annoying at three dimensions and unusable at four.

The issue is that the otype-recursion pattern uses more assertions than needed. For example, array5(**float**)'s default constructor has all four lifecycle assertions about the element type, **float**. However, it only needs **float**'s default constructor, as the other operations are never used. Current work by the C∀ team aims to improve this situation. My workaround moves from otype (value) to

float offers
1 default ctor
2 copy ctor
3 assignment
4 dtor

array5(**float**) has
1 default ctor, using **float**'s
    1 default ctor
    2 copy ctor
    3 assignment
    4 dtor
2 copy ctor, using **float**'s
    1 default ctor
    2 copy ctor
    3 assignment
    4 dtor
3 assignment, using **float**'s
    1 default ctor
    2 copy ctor
    3 assignment
    4 dtor
4 dtor, using **float**'s
    1 default ctor
    2 copy ctor
    3 assignment
    4 dtor

array5(array5(**float**)) has
1 default ctor, using array5(**float**)
    1 default ctor, using **float**'s
        1 default ctor
        2 copy ctor
        3 assignment
        4 dtor
    2 copy ctor, using **float**'s
        1 default ctor
        2 copy ctor
        3 assignment
        4 dtor
    3 assignment, using **float**'s
        1 default ctor
        2 copy ctor
        3 assignment
        4 dtor
    4 dtor, using **float**'s
        1 default ctor
        2 copy ctor
        3 assignment
        4 dtor
2 copy ctor, using array5(**float**)
    ... 4 children, 16 leaves
3 assignment, using array5(**float**)
    ... 4 children, 16 leaves
4 dtor, using array5(**float**)
    ... 4 children, 16 leaves
(64 leaves)

Figure 3.11: Exponential thunk generation under the otype-recursion pattern. Each time one type's function (*e.g.*, ctor) uses another type's function, the C∀ compiler generates a thunk, to capture the used function's dependencies, presented according to the using function's need. So, each non-leaf line represents a generated thunk and every line represents a search request for the resolver to find a satisfying function.

dtype (pointer) with a default-constructor assertion, where dtype does not generate any constructors but the assertion claws back the default otype constructor.

```
// autogenerated for otype−recursion:
forall( T )
void ?{}( array5( T ) & this ) {
    for ( i; 5 ) { ( this[i] ){}; }
}
forall( T )
void ?{}( array5( T ) & this, array5( T ) & src ) {
    for ( i; 5 ) { ( this[i] ){ src[i] }; }
}
forall( T )
void ^?{}( array5( T ) & this ) {
    for ( i; 5 ) { ^( this[i] ){}; }
}
```

```
// lean, bespoke:
forall( T* | { void ?{}( T & ); } )
void ?{}( array5( T ) & this ) {
    for ( i; 5 ) { ( this[i] ){}; }
}
forall( T* | { void ?{}( T &, T ); } )
void ?{}( array5( T ) & this, array5( T ) & src ) {
    for ( i; 5 ) { ( this[i] ){ src[i] }; }
}
forall( T* | { void ?{}( T & ); } )
void ^?{}( array5(T) & this ) {
    for (i; 5) { ^( this[i] ){}; }
}
```

Moreover, the assignment operator is skipped, to avoid hitting a lingering growth case. Temporarily skipping assignment is tolerable because array assignment is not a common operation. With this solution, the critical lifecycle functions are available, with linear growth in thunk creation for the default constructor.

Finally, the intuition that a programmer using an array always wants the elements' default constructor called *automatically* is simplistic. Arrays exist to store different values at each position. So, array initialization needs to let the programmer provide different constructor arguments to each element.

```
thread worker { int id; };
void ?{}( worker & ) = void;          // remove default constructor
void ?{}( worker &, int id );
array( worker, 5 ) ws = {};           // rejected; but desire is for no initialization yet
for ( i; 5 ) (ws[i]){ i };            // explicit (placement) initialization for each thread, giving id
```

Note the use of the C∀ explicit constructor call, analogous to C++'s placement-new. This call is where initialization is desired, and not at the declaration of ws. The attempt to initialize from nothing (equivalent to dropping = {} altogether) is invalid because the worker type removes the default constructor. The worker type is designed this way to work with the threading system. A thread type forks a thread at the end of each constructor and joins with it at the start of each destructor. But a worker cannot begin its forked-thread work without knowing its id. Therefore, there is a conflict between the implicit actions of the builtin **thread** type and a user's desire to defer these actions.

Another C∀ feature for providing C backwards compatibility seems, at first, viable for initializing the array ws; on closer inspection, it is not. C initialization without a constructor is specified with @=, *e.g.*, array(worker, 5) ws @= {} ignores all C∀ lifecycle management and uses C empty initialization. This option does achieve the desired semantics on the construction side. But on the destruction side, the desired semantics is for implicit destructor calls to continue, to keep the join operation tied to lexical scope. C initialization disables *all* implicit lifecycle management, but the goal is to disable only the implicit construction.

To fix this problem, I enhanced the C∀ standard library to provide the missing semantics, available in either form:

```
array( uninit(worker), 5 ) ws1;
array( worker, 5) ws2 = { delay_init };
```

Both cause the ws-construction-time implicit call chain to stop before reaching a worker constructor, while leaving the implicit destruction calls intact. Two forms are available, to parallel the development of this feature in μC++ [8]. Originally μC++ offered only the ws1 form, using the class-template uNoCtor equivalent to C$\forall$'s uninit. More recently, μC++ was extended with the declaration macro, uArray, with usage similar to the ws2 case. Based on experience piloting uArray as a replacement of uNoCtor, it should be possible to remove the first option.

3.8 Array Comparison

3.8.1 Rust

A Rust array is a set of mutable or immutable (**const**) contiguous objects of the same type T, where the compile-time dimension(s) is part of the type signature [T; dimension].

```
let val = 42;
let mut ia: [usize; 5] = [val; 5];          // mutable, repeated initialization [42, 42, 42, 42, 42]
let fval = 42.2;
let fa: [f64; 5] = [1.2, fval, 5.6, 7.3, 9.1];      // immutable, individual initialization
```

Initialization is required even if an array is subsequently initialized. Rust arrays are not VLAs, but the compile-time length can be queried using member len(). Arrays can be assigned and passed to parameters by value or reference, if and only if, the type and dimension match, meaning a different function is needed for each array size.

Array iteration is by a range or array variable.

```
for i in 0..ia.len() { print!("{:?}  ", ia[i] ); }  // 42 42 42 42 42
for fv in fa { print!("{:?}  ", fv ); }              // 1.2 42.2 5.6 7.3 9.1
for i in 0..=1 { ia[i] = i; }                       // mutable changes
for iv in ia { print!("{:?}  ", iv ); }             // 0 1 42 42 42
```

An array can be assigned and printed as a set.

```
ia = [5; 5];   println!( "{:?}  {:?}", ia, ia.len() );  // [5, 5, 5, 5, 5] 5
ia = [1, 2, 3, 4, 5];   println!( "{:?}  {:?}", ia, ia.len() );  // [1, 2, 3, 4, 5] 5
```

Multi-dimensional arrays use nested dimensions, resulting in columns before rows. Here the outer array is a length ROWS array whose elements are f64 arrays of length COLS each.

```
const ROWS: usize = 4;   const COLS: usize = 6;                 [ 0.0, 1.0, 2.0, 3.0, 4.0, 5.0 ]
let mut fmatrix: [[f64; COLS]; ROWS] = [[0.; COLS]; ROWS];      [ 1.0, 2.0, 3.0, 4.0, 5.0, 6.0 ]
for r in 0 .. ROWS {                                            [ 2.0, 3.0, 4.0, 5.0, 6.0, 7.0 ]
    for c in 0 .. COLS {  fmatrix[r][c] = r as f64 + c as f64;  } }   [ 3.0, 4.0, 5.0, 6.0, 7.0, 8.0 ]
```

Slices borrow a section of an array (subarray) and have type &[T]. A slice has a dynamic size and does not implicitly coerce to an array.

```
println!( "{:?}", &ia[0 .. 3] );                    // fixed bounds, [1, 2, 3]
println!( "{:?}", &ia[ia.len() - 2 .. ia.len()] ); // variable bounds, [4, 5]
println!( "{:?}", &ia[ia.len() - 2 .. ] );         // drop upper bound, [4, 5]
println!( "{:?}", &ia[.. ia.len() - 2 ] );         // drop lower bound, [1, 2, 3]
println!( "{:?}", &ia[..] );                       // drop both bound, [1, 2, 3, 4, 5]
```

A multi-dimensional slice can only borrow subrows because a slice requires contiguous memory. Here columns 2–4 are sliced from row 3.

```
let mut slice2: &[f64] = &fmatrix[3][2 ..= 4];
println!( "{:?}", slice2 );                    // [5.0, 6.0, 7.0]
```

Here row 2 is sliced from the sub-matrix formed from rows 1–3.

```
slice2 = &fmatrix[1 ..= 3][1];
println!( "{:?}", slice2 );                    // [3.0, 4.0, 5.0, 6.0, 7.0, 8.0]
```

A slice can be explicitly converted into an array or reference to an array.

```
let slice: &[f64];
slice = &fa[0 ..= 1];                         // create slice, [1.2, 42.2]
let mut fa2: [f64; 2] = <[f64; 2]>::try_from( slice ).unwrap(); // convert slice to array, [1.2, 42.2]
fa2 = (& fa[2 ..= 3]).try_into().unwrap();    // convert slice to array, [5.6, 7.3]
```

The vec macro (using Vec type) provides dynamically-size dynamically-growable arrays of arrays only using the heap (C++ vector type).

Rust	C++
`let (rows, cols) = (4, 6);`	`int rows = 4, cols = 6;`
`let mut matrix = vec![vec![0; cols]; rows];`	`vector<vector<int>> matrix( rows, vector<int>(cols) );`
`... matrix[r][c] = r + c; // fill matrix`	`... matrix[r][c] = r + c; // fill matrix`

A dynamically sized array is sliced the same as a fixed-size one.

```
let mut slice3: &[usize] = &matrix[3][2 ..= 4]; // [5, 6, 7]
slice3 = &matrix[1 ..= 3][1];                 // [2, 3, 4, 5, 6, 7]
```

Finally, to mitigate the restriction of matching array dimensions between argument and parameter types, it is possible for a parameter to be a slice.

```
fn zero( arr: & mut [usize] ){
    for i in 0 .. arr.len() { arr[i] = 0; }
}
zero( & mut ia[0..5] ); // arbitrary sized slice
zero( & mut ia[0..3] );
```

Or write a *generic* function taking an array of fixed-element type and any size.

```
fn aprint<const N: usize>( arg: [usize; N] ) {
    for r in 0 .. arg.len() {   print!( "{}  ", arg[r] );   }
}
aprint( ia );
```

The critical differences between these declaration forms are whether the length can be dynamic and whether multiple occurrences of the same length are expressible. The parameter of zero, whose type, & mut [usize], is a slice, takes both pointer and length at runtime. The parameter of aprint, whose type, [usize; N] is an array, takes only a pointer at runtime, while the function itself is monomorphized (to a specific N value) during compilation. So a zero-like slice can take an unknown length at runtime, while an aprint-like array cannot. But several zero-like slices cannot require that that their sizes correspond (as in C∀'s zero-overhead matrix-multiplication example of Section 3.6, p. 58), while several arrays can. Here therefore, Rust, like others, forces a choice between the two features of dynamic size and agreeing sizes, while the C∀ solution provides both features simultaneously.

Rust offers no native representation of a contiguous, higher dimensional, dynamically sized array. The only dynamically sized option is Vec. Vec is a pointer to a separate (managed) heap allocation. So, a Vec-based matrix is non-contiguous and potentially non-rectangular. This status is precisely the one that Section 3.6, p. 58 builds in C++, and demonstrates is prone to difficulty with bound-check removal.

3.8.2 Java

Java arrays are references, so multi-dimension arrays are arrays-of-arrays (see Section 2.3.3, p. 16). For each array, Java implicitly stores the array dimension in a descriptor, supporting array length, subscript checking, and allowing dynamically-sized array-parameter declarations.

```
   C    void f( size_t n, size_t m, float x[n][m] );
Java    void f( float x[][] );
```

However, in the C prototype, the parameters n and m are manually managed, *i.e.*, there is no guarantee they match with the argument matrix for parameter x. Figure 3.12 compares a triangular matrix (array-of-arrays) in dynamically safe Java to unsafe C. Each dynamically sized row in Java stores its dimension, while C requires the programmer to manage these sizes explicitly (rlnth). All subscripting is Java has bounds checking, while C has none. Both Java and C require explicit null checking, otherwise there is a runtime failure.

The downside of the arrays-of-arrays approach is performance due to pointer chasing versus pointer arithmetic for a contiguous array. Furthermore, there is the cost of managing the implicit array descriptor. It is unlikely that a JIT can dynamically rewrite an arrays-of-arrays form into a contiguous form.

In comparison, Java provides the least support of the features being discussed. It is limited to a noncontiguous representation of multidimensional arrays, while all of Rust, C++ and C∀ offer contiguous, at least in some cases. In the area where C∀ claims an advantage over Rust and C++, Java simply does not compete. Rust and C++ let the user choose one of having a type-system managed length (so that a multitude of arrays of corresponding lengths is expressible), or having a runtime determined length. C∀ supports both abilities simultaneously. Java never supports the first ability; all assertions of length correspondence are explicit runtime checks, and thereby, less accessible for optimization.

Java's object orientation provides a convenient way to express container generalization, *e.g.*, an ArrayList and a LinkedList (both specific) each being a List (general). Conformance at this level is not yet a design goal for C∀ collections. But the same sort of relationship arose in Section 3.5, p. 53, where a function that prints any 1-dimensional array-like collection of numbers (type C, given ar(C, N, **float**) for some N; general) is equally happy working on either a contiguous array-proper, or a noncontiguous slice of a higher-dimensional array, like a column of a row-major matrix (both specific). A comparison of these two mechanisms helps clarify C∀'s status as a non-object-oriented language.

```
int m[][] = {  // triangular matrix
    new int [4],
    new int [3],
    new int [2],
    new int [1],
    null
};

for ( int r = 0; r < m.length; r += 1 ) {
    if ( m[r] == null ) continue;
    for ( int c = 0; c < m[r].length; c += 1 ) {
        m[r][c] = c + r; // subscript checking
    }

}

for ( int r = 0; r < m.length; r += 1 ) {
    if ( m[r] == null ) {
        System.out.println( "null row" );
        continue;
    }
    for ( int c = 0; c < m[r].length; c += 1 ) {
        System.out.print( m[r][c] + "  " );
    }
    System.out.println();

}
```

(a) Java

```
int * m[5] = {  // triangular matrix
    calloc( 4, sizeof(int) ),
    calloc( 3, sizeof(int) ),
    calloc( 2, sizeof(int) ),
    calloc( 1, sizeof(int) ),
    NULL
};
int rlnth = 4;
for ( int r = 0; r < 5; r += 1 ) {
    if ( m[r] == NULL ) continue;
    for ( int c = 0; c < rlnth; c += 1 ) {
        m[r][c] = c + r; // no subscript checking
    }
    rlnth -= 1;
}
rlnth = 4;
for ( int r = 0; r < 5; r += 1 ) {
    if ( m[r] == NULL ) {
        printf( "null row\n" );
        continue;
    }
    for ( int c = 0; c < rlnth; c += 1 ) {
        printf( "%d ", m[r][c] );
    }
    printf( "\n" );
    rlnth -= 1;
}
```

(b) C

Figure 3.12: Triangular matrix in Java and C

In object-oriented polymorphism, an abstract operation like List.add corresponds with a slot in a virtual table (vtable)[8]. The vtable is defined on the abstract type List, codifying its guarantee, "A List is a thing that supports adding an element," as, "At vtable slot 17, will be a function pointer for the appropriate add."

Of all the general/specific permutations, ArrayList< List< **float** > > is one that shows the vtable design consequences well. Consider looping through one of these outer ArrayList objects. At the current position, a List< **float** > is retrieved. A get-first-element call occurs on it. So, the current item's vtable is accessed, then the function pointer from slot 17, and that function is called. The call goes to LinkedList's get-first-element because the current item is LinkedList.

[8]This abbreviated presentation treats List as if it were a singly-inherited class, rather than introducing the further complexity of List being a multiply-inherited interface. The C∀ comparison is unaffected; in reality, dispatch by Java List incurs a futher level of lookup/indirection, which is incidental to the comparison. Optimizations that can remove both base-class and interface method lookup costs from a program's hot path are also not discussed.

Consider the same story playing out with C∀ trait polymorphism. The type, concrete wrapping abstract, is that of the parameter to f in:

```
forall( [N_outr], C_inr, [N_inr] | ar( C_inr, N_inr, float ) )
void f( array( C_inr, N_outr ) & );
```

And the actual type, everything-concrete is

```
array( dlistv( float ), n );
```

where dlistv is a hypothetical wrapped list offering a C++-style value-only interface (see Figure 2.5, p. 27). Though an eventual conformed collections' design would not have it so, suppose dlistv implements ar, the only general-collection type in the present C∀ vocabulary. Consider looping through one of these outer array objects. At the current position, a C_inr is retrieved. A get-first-element call occurs on it. So, the polymorphic type information of C_inr (received by implicit parameter) applies, provides the function pointer and the call to that function occurs. The call goes to dlistv's get-first-element because C_inr is dlistv.

The difference shows up at the next outer iteration. At this subsequent position, the element comes, again, as the abstract type List< **float** >/C_inr. The steps proceed as before: C++ asks the element for its get-first-element function; C∀ uses the get-first-element function that it received as a parameter. In C++, this subsequent item happens to be ArrayList, and the vtable provides arrayList's function. In C∀, this outcome is not possible; at the function call with the outer loop, a single get-first-element function is provided, which applies to all elements.

In the Java version, each entry in the outer ArrayList carries its own identity, revealing itself to be ArrayList, LinkedList, or potentially anything else, only after the item is encountered during traversal. And even there, the revelation is not some, 'I am T," but only implied by which function receives the subscript call. In the C∀ version, C_inr is fixed at the f invocation.

So, C∀'s polymorphism most naturally provides for homogenous collections, while Java's naturally provides for heterogeneous.

One is not better than the other; hybrid arrangements apply in practice. The C∀ project is exploring ways to support heterogeneous dispatch. Java offers ways for programmers to restrict just how different two objects of the same compile-time type can be. A Java runtime (including its hardware) includes multiple optimizations for the frequent case when a potentially heterogeneous situation ends up running uniformly.

Homogeneity is necessary for putting items directly into an array. Java's heterogeneous-by-default collections, even array, work because the collection is only able to store a pointer. This necessary pointer indirection means even array-of-array is non-contiguous. With the inner collections non-contiguous, they might as well have different lengths.

C∀, on the other hand. lets the programmer lock things down, precluding such possibilities. The type of *all* the inner collections is fixed by a parameter given once. That the element type includes the length is, of course, a key contributor to, "It's a rectangle." But, more essentially to the Java-C∀ polymorphism difference, this upfront commitment to a *concrete* type precludes situations where a certain entry partway through wishes to claim to be a subtype that needs extra storage fields. All elements commit to being exactly the same size; so, the elements fit directly into an array, without pointer indirection. Thereby, truly contiguous storage is made possible.

3.8.3 C++

Because C arrays are difficult and dangerous, the mantra for C++ programmers is to use std::vector in place of the C array. While the vector size can grow and shrink dynamically, *vs.* an unchanging dynamic size with VLAs, the cost of this extra feature is mitigated by preallocating a known size (like the VLA) at the declaration. So, it costs one upfront dynamic allocation and avoids growing the array through pushing.

```
vector< vector< int > > m( 5, vector<int>(8) ); // initialize size of 5 x 8 with 6 dynamic allocations
```

Multidimensional arrays are arrays-of-arrays with associated costs. Each vector array has an array descriptor containing the dimension, which allows bound checks using x.at(i) in place of x[i]. Used with these restrictions, out-of-bound accesses are caught, and in-bound accesses never exercise the vector's ability to grow, preventing costly reallocate-and-copy, and never invalidating references to contained values.

This scheme matches a Java array's safety and expressiveness exactly, with the same inherent costs. Notably, a C++ vector of vectors does not provide contiguous element storage (even when upfront allocation is done carefully) because a vector puts its elements in an auxiliary allocation. So, in spite of vector< vector< int > > appearing to be "everything by value," it is still an allocated array whose elements include pointers to further arrays.

C++ has options for working with memory-adjacent data in desirable ways, particularly in recent revisions. But none makes an allocation with a dynamically given fixed size less awkward and performance-limited than the vector arrangement just described.

C++ 26 adds std::inplace_vector providing an interesting vector–array hybrid. Like a vector, it lets a user construct elements in a loop, rather than imposing uniform construction. Yet it preserves std::array's ability to be entirely stack-allocated, by avoiding an auxiliary elements' allocation. However, it preserves the fundamental limit of std::array, that the length, being a template parameter, must be a static value.

C++ 20's std::span is a view that unifies true arrays, vectors, static sizes and dynamic sizes, under a common API that adds bounds checking. When wrapping a vector, bounds checking occurs on regular subscripting, *i.e.*, one need not remember to use .at. When wrapping a locally declared fixed-size array, bound communication is implicit. But it has a soundness gap by offering construction from pointer and user-given length.

C++ 23's std::mdspan adds multidimensional indexing and reshaping capabilities analogous to those built into C∀'s array. Like span, it works over multiple underlying collection types. But neither span nor mdspan augments the available allocation options.

Thus, these options do not offer an allocation with a dynamically given fixed size. And furthermore, they do not provide any improvement to the C Flexible Array Member pattern, for making a dynamic amount of storage contiguous with its header, as do C∀'s new Dynamic Array Members.

The options of std::array, std::inplace_vector and some instantiations of std::span and std::mdspan give type-system managed statically-fixed lengths. The options of std::vector and the remaining instantiations of std::span and std::mdspan give type-system unmanaged dynamic lengths. The C∀ array gives both abilities simultaneously: type-system managed dynamic (and optionally static) lengths.

3.8.4 Subscript *vs*. iteration

Most programming languages support an abstract access pattern for their collections based on iterators. This design feature most prominently speaks to, "How is an array like a linked list, or sequence read from a pipe?" But, more subtly, and entirely within the array realm, the pattern also clarifies which of the fundamental array abilities an application is using: cache-optimal sequential access *vs*. random access.

Rust is openly critical of unnecessary array indexing, *i.e*., of making sequential access look like potentially random access. Its built-in linter warns (by default) when it finds this pattern. The following iterator examples use Rust, though rather similar facilities exist in Java, JavaScript, C++, C#, Swift and Python. Rust is the most complete in ensuring that mutation works through the iterator abstraction; many of the others fall back on index-based access for mutation.

An iterator-based loop is:

```
for row in &mut matrix {
    for cell in row { // mut implied
        *cell += 42;
    }
}
```

When an index is also needed, the iterator algebra provides it.

```
for (i, row) in matrix.iter_mut().enumerate() {
   for (j, cell) in row.iter_mut().enumerate() {
        *cell = i * j;
   }
}
```

An advantage of this style is that bound checking is is unnecessary, accesses being in-bound by construction.

So this iterator looping pattern can be seen as an alternative to the looping of Section 3.6, p. 58. There, C∀ uses implicitly, correctly, communicated bounds as the basis of looping over valid indices. Rust seems to have a different solution to the same problem: get rid of the problematic indices entirely.

C∀ should, some day, offer composable, conformed iterators over its collections. These iterators should enable the Rust-style no-index access. It is important for a language to help simple cases look simple.

However, no-index access is strictly less expressive than index-based access. Rust's iterator composition struggles to handle the key example of Section 3.6, p. 58, dot-product formulated matrix multiplication, shown in Figure 3.13. Here, the i and k loops are rewritten, but j, which picks a column from the right-hand parameter, is left traditional, because a column is defined positionally, as, "The j-th element of every row;" something has to say which position.

Moreover, cases with essential indexing include in-place swaps and partitioning (sorting, reversing) and dynamic programming tables where memo[i] depends on memo[f(i)].

So, iterator-style element access helps keep simple cases looking simple, but the ability to judge a[i] being in-bound, given **forall**([N]) ... array(T, N) ... i < N, is an indispensable general case.

```
fn matmul(l: &[Vec<f64>], r: &[Vec<f64>]) -> Vec<Vec<f64>> {
    let n = r.first().map_or(0, |r| r.len());
    l.iter()
        .map(|l_row| {
            let mut out = Vec::with_capacity(n);
            for j in 0..n {
                let col = r.iter().map(|r_row| r_row[j]);  // column j
                out.push(l_row.iter().zip(col).map(|(x, y)| x * y).sum::<f64>());
            }
            out
        })
        .collect()
}
```

Figure 3.13: Mostly iterator-based matrix multiplication in Rust

3.9 Future Work

The ultimate goal is to replace all C arrays with C∀ arrays, establishing a higher level of safety in C programs. To that end, this section surveys possible next steps that could support this shift.

3.9.1 Array Syntax

An important goal is to change the C∀ array syntax, away from the current array(...), into C-style "real array" syntax. The proposal (which is partially implemented) is an alternate dimension and subscript syntax for multi-dimension arrays. C multi-dimension and subscripting syntax uses multiple bracketed constants/expressions.

```
int m[2][3];  // dimension
m[0][1] = 3;  // subscript
```

The alternative C∀ syntax is a comma separated list:

```
int m[2, 3];  // dimension
m[0, 1] = 3;  // subscript
```

which should be intuitive to C programmers and is used in mathematics $M_{i,j}$ and other programming languages, *e.g.*, PL/I, Fortran. With respect to the dimension expressions, C only allows an assignment expression, not a comma expression.

```
    a[i, j];
test.c:3:16: error: expected ']' before ',' token
```

However, there is an ambiguity for a single dimension array, where the syntax for C and C∀ arrays are the same, **int** ar[10]. A solution is to use a terminating comma to denote a C∀-style single-dimension array.

```
int ar[2,];  // single dimension CFA array
```

This syntactic form is also used for the (rare) singleton tuple [y,]. The extra comma in the dimension is only mildly annoying, and acts as a clear marker to differentiate C and C∀ arrays. Hence, C∀

can repurpose the comma expression in this context for a list of dimensions. Should the C/C∀ array distinction disappear in the future the need for the terminating comma would too. With respect to subscript expressions, C allows a comma expression here. However, a comma expression in this context is rare, and is most commonly a (silent) mistake: subscripting a C matrix with `m[i, j]` instead of `m[i][j]` selects the jth row not the i, j element. It is still possible to write `m[(i, j)]` in the new syntax to achieve the equivalent of the old `m[i, j]`. Internally, the compiler must desugar `[i, j, k]` into `[i][j][k]` to match with three calls to subscript operators. There is no ambiguity for subscripting a single dimensional array, as the subscript operator selects the correct form from the array type. Currently, `array` supports the old and new subscript syntax (see Figure 3.9, p. 62), including combinations of new and old, `arr[1, 2][3]`. The new subscript syntax can be extended to C arrays for uniformity, but requires the non-compatible removal of the (rare) comma-expression as a subscript. Commas also occur in users' macro calls, but these are necessarily parenthesized, so there is no ambiguity.

3.9.2 Error Messages

Length was a first foray into non-type polymorphic parameters. In one respect, this work's solution is less general than C++'s non-type template parameters[9], which can accept strings, functions and values of many more types. A **forall**([N]) parameter is `size_t`, only.

It would be a significant change to generalize from C∀'s present solution to C++-equivalent type coverage. The C∀ compiler's resolver, the most complex piece, would need to be upgraded to understand these non-type parameters directly.

Today's solution leverages a special relationship between types and `size_t`: the two-way mapping afforded by **char**[n] and **sizeof**(T). This mapping allows for desugaring user input into the language that the pre-existing resolver understands, as detailed in Section 3.3, p. 45.

The downside of this desugaring, as Section 2.4.5, p. 30 admonishes, is that it does not provide a means for reporting a user-caused error, from its discovery in the resolver, to the user's attention, without including clutter from the desugaring. The C∀ team generally recognizes substantial room for improvement in even general, resolver-alone, errors. I fully acknowledge that the current state of errors from C∀ array usage mistakes makes the C∀ array system hard to learn.

3.9.3 Retire Pointer Arithmetic

C∀ has a tremendously flexible core machinery, in which facts about C are expressed. Even the fact that there is a `?+?` operator for a pair of **int**s is not baked into the compiler proper; it is expressed, as a C∀ source-code declaration, by an implicit pseudo-include, in the *prelude*. More controversial facts are stated there too, notably, that every pointer type has arithmetic and subscript operators. As Section 2.3.1, p. 12 explains, this fact is a necessary component in how a *C array* is indexed. But this fact is unused when dereferencing a C∀ `array`, as the subscript operator, in this case, is defined directly on `array`, and not via an intermediate pointer type.

I conducted a successful proof of concept for the technical elements that underpin this vision.

- Demote pointer-arithmetic operators from permanent prelude members to header-includable.

[9] In another respect, use of dynamic values, the C∀ solution can do more.

- Include them when in a C-compatibility mode, but not when in an enhanced-safety mode.
- Let a development team who maintains a compile unit that they believe they have onboarded to use only new C∀ arrays, set it to build in enhanced-safety mode.
- When a developer falls back on habit and forgetfully uses a C array in the onboarded module, the compiler's error serves as a reminder and a protector of the onboarded code quality.

This notion should be explored beyond the toy examples it has seen to date.

3.9.4 Defer Size Tracking

The current length checking and communication works well in the style of example illustrated in this thesis. The characteristic of this style is:

- Alice holds an array and knows its size.
- Bob offers a service that can apply to an array of any size.
- Alice calls Bob, passing her array; Bob (implicitly) learns the array's (correct) size.

This interaction is straightforward universal polymorphism, where Bob is parameterized by size. And compared with the following alternative, it is much more common.

But size knowledge could just as well flow the other way:

- Bob offers a service that provides access to an array of a specific size known to Bob.
- Alice is content to work on an array of any size.
- Alice calls Bob, and receives his array as a return; Alice (implicitly) learns the array's (correct) size.

This interaction is the dual notion of existential polymorphism, working upon size information. Work [22] accompanying the array-centric language Futhark has detailed its theory.

An existential size helps cope with unnecessary repetition that occurs as program size increases and more intermediaries separate an array producer and consumer. Consider the School structure of Figure 3.3, p. 44, and imagine passing one through a chain of intermediaries. While the producer and consumer have every reason to know the school's sizes, the intermediaries do not. Yet, they too would have to be parameterized by the numbers of courses and students, so that they could have them in scope at the point where they name the school's type.

The desired solution is to write the size down, somewhere in relation to the school on which producer and consumer agree, but where it need not bother the intermediaries.

```
struct SchoolRef {
    size_t classes;
    size_t students;
    School(classes, students) * school;
};
```

This declaration is wishful; at the school member declaration site, the names classes and students are not actually in scope. But if it worked, SchoolRef would be doing the intermediaries a tremendous service. It is not a polymorphic type. The intermediaries no longer need size information in scope to pass one along. SchoolRef stops the polymorphism of School from flowing farther outward. The final consumer need only open the box, discover what is inside, and proceed.

```
forall( [C], [S] ) static void process( School(C, S) * );
void receive( SchoolRef r ) {
    with( r ) {
        classes; students;                          // are in scope
        School( classes, students ) * s2 = school; // state the type
        process( s2 );                              // communicate the type
    }
}
```

3.9.5 Performance Benchmarking and Case Studies

Section 3.6, p. 58 infers that C∀ array performance is acceptable because the C∀ array erases to a C array. This assurance is equivalent to the microbenchmark experiments presented for linked lists (Section 4.6, p. 89) and strings (5.4, p. 131).

An assurance of a more applied quality would be obtained by running community-accepted full benchmarks, featuring code sourced from the wild. As for the other collection types, this level of experimentation was not undertaken within this thesis because no free upgrade path exists. Real effort is required to convert from-the-wild C array code into C∀ array code. This effort comes from the C∀ array system first challenging users to state *higher* bars for correctness than allowed by C arrays, and then requiring users to meet these bars to the type system's satisfaction.

Still, such an assessment is worthwhile. It would, moreover, serve as an assessment of the C∀ array system's usability.

The array is an attractive first candidate for a real-benchmark assessment because there is abundant well-measured C code that uses arrays heavily. Likely half the SPEC CPU suite qualifies.

By comparison, to test C∀ string *vs.* **char** * requires finding code that is a known heavy <string.h> user, unlikely in a general SPEC program. And to test C∀ string *vs.* C++ string requires turning the rest of the benchmark's C++ program into C∀.

One aspect of the methodology of Section 3.6, p. 58 is recommended to carry forward with real benchmarking. Leveraging C as the common pseudo-subset of both C++ and C∀ enables a two-pronged conversion. Upgrade the original code both to the C∀ array and to C++ vector with .at checked access. Then, in situations where a bound is reused across multiple arrays, there is a hope/hypothesis that C∀ delivers better performance.

3.9.6 Guarantee Zero Overhead

Section 3.6, p. 58 demonstrates how C∀ does well at skipping unnecessary runtime bound checks. Nevertheless, the scheme presented puts a programmer's hopes in the hands of the optimizer. The programmer writes code in which it appears that a bound check ought to be unnecessary, but feedback about success or failure there is indirect.

Dex [42] is a contemporary array-centric language that follows in the Ada array tradition [1] of designating a type to be used for subscripting an array. While C∀ has its new [N] living in the type system, this N is used to represent an array's size. An indexing type would be the type of i, in ar[i].

```
const size_t n = ask_the_user();
typedef range(0, n) till_n_t;                   // does not exist in CFA yet
till_n_t x = 42;                                // static reject
till_n_t y = check(42);                         // static accept, dynamic check
void f( array(int, n) & ar, till_n_t i ) {
    sout | ar[ i ];                             // bound check unnecessary
}
```

In today's C∀ solution, any such i is size_t.

```
void f2( array(int, n) & ar, size_t i ) {
    sout | ar[ i ];                             // bound check necessary
}
```

With a constrained index type, bound checks shift from being implicit at the time of an array access, to explicit at the time of a conversion. A user then has the option to import only the index-typed subscript operator. In such a setup, no implicit bound check can occur. Bound checks occur exactly where the user places them. If the user succeeds at needing no explicit bound checks, then the user is assured that there are no bound checks.

Prior C∀ work [35] has touched upon this topic. Therein, a user's enumeration type is adapted to serve as the domain of an array. However, this work progressed as far as inferring an array size from the enumeration type, *e.g.*, that **int** happy_hours[weekday_t] should be a declaration with length seven. It did not address forbidding happy_hours[42].

Both enumerations and the above hand-waived range(0, n) are valid indexing types. Supporting indexing types would put the user more in control of a program's performance and failure modes.

Chapter 4

Linked List

This chapter presents my work on designing and building a linked-list library for C∀. Due to time limitations and the needs expressed by the C∀ runtime developers, I focussed on providing a doubly linked list and its bidirectional iterators for traversal. Simpler data-structures, like stack and queue, can be built from the doubly-linked mechanism with only a slight storage/performance cost because of the unused link field. Reducing to data-structures with a single link follows directly from the more complex double links and its iterators.

4.1 Plan-9 Inheritance

This chapter uses a form of inheritance from the Plan-9 C dialect [53, § 3.3], which is supported by gcc and clang using −fplan9−extensions. C∀ has its own variation of the Plan-9 mechanism, where the nested type is denoted using **inline**.

```
union U {  int x;  double y;  char z; };
struct W { int i;  double j;  char k; };
struct S {
    inline struct W;                    // extended Plan-9 inheritance
    unsigned int tag;
    inline U;                           // extended Plan-9 inheritance
} s;
```

Inline inheritance is containment, where the inlined field is unnamed and the type's internal fields are hoisted into the containing structure. Hence, the field names must be unique. The position of the containment is normally unimportant, unless there is some form of memory or **union** overlay. Finally, the inheritance declaration of U is not prefixed with **union**. Like C++, C∀ allows optional prefixing of type names with their kind, *e.g.*, **struct**, **union**, and **enum**, unless there is ambiguity with variable names in the same scope.

The choice of inlining a type is orthogonal to the choice of nesting that type's declaration.

```
struct part1 { int i; };
struct whole {
    struct part2 { float j; };
    part1 fld1;
    part2 fld2;
    inline part1;
```

```
void f( U, U * );                       // value, pointer
void g( W, W * );                       // value, pointer
U u, * up;   W w, * wp;   S s, * sp;
u = s;   up = sp;                       // value, pointer
w = s;   wp = sp;                       // value, pointer
f( s, &s );                             // value, pointer
g( s, &s );                             // value, pointer
```

Figure 4.1: Plan-9 polymorphism

```
    inline part2;
};
whole w;
w.fld1.i; w.i; w.fld2.j; w.j;

part1 p1;
whole.part2 p2;
```

So, a field can be of any type and a type can be declared nested or not. Inlining is about a field; it is the choice of whether the field should be named (normal) or anonymous (inlined).

Figure 4.1 shows the key polymorphic feature of Plan-9 inheritance: implicit conversion of values and pointers for inherited types. In the example, there are implicit conversions from S to U and S to W, extracting the appropriate value or pointer for the substructure. Figure 4.2 shows complex multiple inheritance patterns are possible, like the *diamond pattern* [50, § 6.1][9, § 4]. While C++ offers two takes on the diamond pattern, simple and virtual, C∀ inline inheritance is simple, with no current support for virtual. In simple inheritance, each path from most-derived type to reused base type defines a further copy of the base type within the derived type. In virtual inheritance, a single copy of the reused base type is accessed by all paths from the most-derived type. In the diamond case that C∀/Plan-9 inheritance enables for the C∀ linked list[1], the simple version is the desired one.

4.2 Features

The following features directed this project, where the goal is high-performance list operations required by C∀ runtime components, like the threading library.

4.2.1 Core Design Issues

The doubly-linked list attaches links intrusively in a node, allows a node to appear on multiple lists (axes) simultaneously, integrates with user code via the type system, treats its ends uniformly, and optionally identifies a list using an explicit head. This design covers system and data management issues stated in Section 2.4.1, p. 24.

[1] See discussion of Figure 4.5, p. 83.

C++

```
struct B { int b; };
struct L : public B { int l, w; };
struct R : public B { int r, w; };
struct T : public L, R { int t; };
                T
          B     L       B     R
T t = { { { 5 }, 3, 4 }, { { 8 }, 6, 7 }, 3 };
t.t = 42;
t.l = 42;
t.r = 42;
((L &)t).b = 42;  // disambiguate
```

C∀

```
struct B { int b; };
struct L { inline B; int l, w; };
struct R { inline B; int r, w; };
struct T { inline L; inline R; int t; };
                T
          B     L       B     R
T t = { { { 5 }, 3, 4 }, { { 8 }, 6, 7 }, 3 };
t.t = 42;
t.l = 42;
t.r = 42;
((L &)t).b = 42;  // disambiguate, proposed solution t.L.b = 42;
```

Figure 4.2: Diamond non-virtual inheritance pattern

```
struct req {
    int pri, rqr;
    inline dlink(req);  // containment inheritance, fields hoisted into structure
};

dlist(req) reqs;

req r1 = {1, 42}, r2 = {2, 42};

insert_first(reqs, r2);
insert_first(reqs, r1);
```

Figure 4.3: Demonstration of the running req example, done using the C∀ list library. This example is equivalent to the three approaches in Figure 2.4, p. 25.

Figure 4.3 continues the running req example from Figure 2.4, p. 25 using the C∀ list dlist. The C∀ link attachment is intrusive so the resulting memory layout is per user node, as for the LQ version of Figure 2.4(a), p. 25. The C∀ framework provides generic type dlink(T), not to be confused with dlist, for the two link fields (forward and backward directions). A user inserts the links into the req structure via C∀ inline-inheritance (see Section 4.1, p. 79). Lists leverage the automatic conversion of a pointer to anonymous inline field for assignments and function calls. Therefore, a reference to a req is implicitly convertible to dlink in both contexts. The links in dlink point at another embedded dlink node, know the offsets of all links (data is abstract but accessible), and any field-offset arithmetic or link-value changes are safe and abstract.

Figure 4.4 shows the outline for types dlink and dlist. Note that the first **forall** clause is distributed across all the declaration in its containing block, eliminating repetition on each declaration. The second nested **forall** on dlist is also distributed and adds an optional second type parameter, tLinks, denoting the linking axis (see Section 2.4.4, p. 26), *i.e.*, the kind of list this node can appear on.

```
forall( tE & ) {                                    // distributed
    struct dlink { ... };                           // abstract type
    static inline void ?{}( dlink( tE ) & this );  // constructor

    forall( tLinks & = dlink( tE ) ) {              // distributed, default type for axis
        struct dlist { ... };                       // abstract type
        static inline void ?{}( dlist( tE, tLinks ) & this ); // constructor
    }
}
```

Figure 4.4: Sketch of the dlink and dlist type definitions

Figure 4.5 shows how the C∀ library supports multi-inline links, so a node has multiple axes. The declaration of req has two inline-inheriting dlink occurrences. The first of these gives a type named req.by_pri; req inherits from it and it inherits from dlink. The second line req.by_rqr is similar to req.by_pri. Thus, there is a diamond, non-virtual, inheritance from req to dlink, with by_pri and by_rqr being the mid-level types (see Figure 4.2).

The declarations of the list-head objects, reqs_pri, reqs_rqr_42, reqs_rqr_17, and reqs_rqr_99, bind which link nodes in req are used by this list. Hence, the type of the variable reqs_pri, dlist(req, req.by_pri), means operations on reqs_pri implicitly select (disambiguate) the correct dlinks, *e.g.*, the calls insert_first(reqs_pri, ...) imply, "here, we are working by priority." As in Figure 2.6, p. 28, three lists are constructed, a priority list containing all nodes, a list with only nodes containing the value 42, and a list with only nodes containing the value 17.

The list library also supports the common case of a single axis more naturally than LQ. Returning to Figure 4.3, the single-axis list has no contrived name for the link axis, as it uses the default type in the definition of dlist. In contrast, the LQ list in Figure 2.4(a), p. 25 adds the unnecessary field name d. In C∀, a single axis list sets up a single inheritance with dlink, and the default list axis is to itself.

When operating on a list with several axes and operations that do not take the list head, the list axis can be ambiguous. For example, a call like insert_after(r1, r2) does not have enough information to know which axes to select implicitly. Is r2 supposed to be the next-priority request after r1, or is r2 supposed to join the same-requester list of r1? As such, the C∀ type-system gives an ambiguity error for this call. There are multiple ways to resolve the ambiguity. The simplest is an explicit cast on each call to select the specific axis, *e.g.*, insert_after((by_pri)r1, r2). However, multiple explicit casts are tedious and error-prone. To mitigate this issue, the list library provides a hook for applying the C∀ language's scoping and priority rules.

```
with ( DLINK_VIA( req, req.pri ) ) insert_after( r1, r2 );
```

Here, the **with** statement opens the scope of the object type for the expression; hence, the DLINK_VIA result causes C∀'s overload resolution to prefer the indicated list axis over other options.

This boost can be applied across multiple statements in a block or an entire function body.

```
with( DLINK_VIA( req, req.pri ) ) {
    ... insert_after( r1, r2 ); ...
}
```

```
void f() with( DLINK_VIA( req, req.pri ) ) {
    ... insert_after( r1, r2 ); ...
}
```

C∀

```
struct req {
    int pri, rqr;
    inline struct by_pri { inline dlink(req); };
    inline struct by_rqr { inline dlink(req); };
};


dlist(req, req.by_pri) reqs_pri;
dlist(req, req.by_rqr) reqs_rqr_42;
dlist(req, req.by_rqr) reqs_rqr_17;
dlist(req, req.by_rqr) reqs_rqr_99;


struct req
    r42a = {1, 42},
    r42b = {2, 42},
    r17a = {2, 17},
    r17b = {3, 17},
    r17c = {4, 17},
    r99a = {3, 99};

insert_first(reqs_pri, r17c);
insert_first(reqs_pri, r99a);
insert_first(reqs_pri, r17b);
insert_first(reqs_pri, r42b);
insert_first(reqs_pri, r17a);
insert_first(reqs_pri, r42a);

insert_first(reqs_rqr_42, r42b);
insert_first(reqs_rqr_42, r42a);

insert_first(reqs_rqr_17, r17c);
insert_first(reqs_rqr_17, r17b);
insert_first(reqs_rqr_17, r17a);

insert_first(reqs_rqr_99, r99a);
```

C

```
struct req {
    int pri, rqr;
    LIST_ENTRY( req ) by_pri;
    LIST_ENTRY( req ) by_rqr;
};

LIST_HEAD( reql, req );

struct reql reqs_pri;
struct reql reqs_rqr_42;
struct reql reqs_rqr_17;
struct reql reqs_rqr_99;

LIST_INIT( &reqs_pri );
LIST_INIT( &reqs_rqr_42 );
LIST_INIT( &reqs_rqr_17 );
LIST_INIT( &reqs_rqr_99 );

struct req
    r42a = {1, 42},
    r42b = {2, 42},
    r17a = {2, 17},
    r17b = {3, 17},
    r17c = {4, 17},
    r99a = {3, 99};

LIST_INSERT_HEAD( &reqs_pri, &r17c, by_pri );
LIST_INSERT_HEAD( &reqs_pri, &r99a, by_pri );
LIST_INSERT_HEAD( &reqs_pri, &r17b, by_pri );
LIST_INSERT_HEAD( &reqs_pri, &r42b, by_pri );
LIST_INSERT_HEAD( &reqs_pri, &r17a, by_pri );
LIST_INSERT_HEAD( &reqs_pri, &r42a, by_pri );

LIST_INSERT_HEAD( &reqs_rqr_42, &r42b, by_rqr );
LIST_INSERT_HEAD( &reqs_rqr_42, &r42a, by_rqr );

LIST_INSERT_HEAD( &reqs_rqr_17, &r17c, by_rqr );
LIST_INSERT_HEAD( &reqs_rqr_17, &r17b, by_rqr );
LIST_INSERT_HEAD( &reqs_rqr_17, &r17a, by_rqr );

LIST_INSERT_HEAD( &reqs_rqr_99, &r99a, by_rqr );
```

Figure 4.5: Demonstration of multiple static link axes in C∀. The right LQ example is repeated from Figure 2.6, p. 28, to ease comparison. The left C∀ example does the same job.

Within the **with**, code that would otherwise be ambiguous acts as if there is only the one list axis, needing no explicit casting.

This boost does not remove the other axes from consideration.

```
insert_last( reqs_pri, r3 ); // req.pri axis implied (discussed earlier)
with ( DLINK_VIA( req, req.rqr ) )
    insert_last( reqs_pri, r4 ); // still req.pri
```

By this technicality, DLINK_VIA does not guarantee that the indicated axis applies always. But it means the indicated axis applies whenever it could apply.

Unlike the C++ template collection types, the CFA library works completely within the type system; both dlink and dlist are ordinary types, not macros subject to expansion. Though C++ template expansion differs from C-family preprocessing, both in being a technically distinct phase of compilation and in having a stricter prerequisite system[2], it remains fundamentally macro-styled. A called template definition may contain references that are (in)valid only by the particulars of template arguments masheing with the template body[3]. The CFA polymorphism system, and so the CFA list, never relies on checking an expanded form. The only such expansion occurs late in compilation, where the optimizer substitutes header-included static-inline functions for performance. Hence, errors in user code are reported only with mention of the library's declarations, versus long template names in error messages.

4.3 List API

The API for the doubly-linked list operations follows. E is the element type, a user-defined structure like req of Figure 4.3, p. 81.

E & listed(E & node);

- returns true if the node is an element of a list and false otherwise

E & empty(dlist(E) & list);

- returns true if the list has no nodes and false otherwise

E & first(dlist(E) & list);

- returns a reference to the first node of the list without removing it or 0p if the list is empty

E & last(dlist(E) & list);

- returns a reference to the last node of the list without removing it or 0p if the list is empty

E & insert_before(E & before, E & node);

- adds a node before a specified node (see insert_last for insertion at the end)[4]

[2] C++-23 concepts improves this system; their adoption in the standard collection library has further mitigated this issue

[3] C++-23 concepts are a system for enabling earlier pre-mash checks, but this system does not restrict against depending on deferred-till-mash checks.

[4] Some list packages allow 0p (nullptr) for the before/after node implying insert/remove at the start/end of the list, respectively. However, this inserts an **if** statement in the fastpath of a potentially commonly used list operation. It is also likely to add the extra parameter demonstrated in 2.4.7, p. 31. In this CFA API, the iter result (below) serves to mark this $n + 1^{\text{st}}$ position.

E & insert_after(E & after, E & node);

- adds a node after a specified node (see insert_first for insertion at the start)

E & remove(E & node);

- removes a specified node from the list (any location) and returns a reference to the node

E & iter(dlist(E) & list);

- creates a non-started iterator for the list; see Figure 4.7, p. 87-(iv)

bool advance(E && refx);

- returns true if the iterator cursor is moved to the next (successor, towards last) node after the prior cursor node and false otherwise

bool recede(E && refx);

- returns true if the iterator cursor is moves to the previous (predecessor, towards first) node before the prior cursor node and false otherwise

bool first(E & node);

- returns true if the node is the first node in the list and false otherwise

bool last(E & node);

- returns true if the node is the last node in the list and false otherwise

E & prev(E & node);

- returns a reference to the previous (predecessor, towards first) node before a specified node or 0p if a specified node is the first node in the list

E & next(E & node);

- returns a reference to the next (successor, towards last) node after a specified node or 0p if a specified node is the last node in the list

E & insert_first(dlist(E) & list, E & node);

- adds a node to the start of the list so it becomes the first node and returns a reference to the node for cascading

E & insert_last(dlist(E) & list, E & node); *// synonym insert*

- adds a node to the end of the list so it becomes the last node and returns returns a reference to the node for cascading

E & remove_first(dlist(E) & list);

- removes the first node and returns a reference to it or 0p if the list is empty

E & remove_last(dlist(E) & list);

- removes the last node and returns a reference to it or 0p if the list is empty

void transfer(dlist(E) & to, dlist(E) & from);

- transfers all nodes from the from list to the end of the to list; the from list is empty after the transfer

void split(dlist(E) & to, dlist(E) & from, E & node);

- transfers the from list up to node to the end of the to list; the from list becomes the list after the node; the node must be in the from list.

```
#include <fstream.hfa>
#include <list.hfa>
struct node {
    int v;
    inline dlink(node);
};
int main() {
    dlist(node) list;
    node n1 = { 1 }, n2 = { 2 }, n3 = { 3 }, n4 = { 4 };
    insert( list, n1, n2, n3, n4 );                    // insert in any order
    sout | nlOff;
    for ( ... ) sout | it.v | ", ";   sout | nl;       // iteration examples plug in here
    remove( n1, n2, n3, n4 );                          // remove in any order
}
```

Figure 4.6: Iterator driver. Specific loop headers from Figure 4.7 plug in, to replace the identified stub loop header.

For operations insert_*, insert, and remove, a variable-sized list of nodes can be specified using C∀'s tuple type [39, § 4.7] (not discussed), *e.g.*, insert(list, n1, n2, n3), recursively invokes insert(list, n_i).[5]

4.3.1 Iteration

It is possible to iterate through a list manually or using a set of standard macros. Figure 4.6 shows the iterator outline, managing a list of nodes, used throughout the following iteration examples. Each example uses this loop body to print the value in the current node.

Figure 4.7-(i–iii) gives a manual iteration method that is low-level but allows complete control of the iteration. The list cursor (index) can be either a pointer or a reference to a node in the list. The choice depends on how the programmer wants to access the fields: it–>f or it.f. The following examples use a reference because the loop body manipulates the node values rather than the list pointers. The end of iteration is denoted by the loop cursor returning 0p.

Figure 4.7-(iv) gives a shorthand for consuming the entire list. Here, the loop variable, it, is *initialized* to an invalid position by = iter(list), then made valid by a first successful call to advance or recede. These functions return boolean. This usage mimics the C++ idiom: **int** i; **while** (cin >> i). Note that this pattern, by virtue of C∀'s syntax for declaring a while-loop variable, implies that the advance/recede call is the *only* code that sees the loop variable in the invalid position.

There are convenience macros that look like foreach in other programming languages. Figure 4.7-(v–vi) shows their application. Macros are not ideal, so future work is to provide a language-level foreach statement, like C++.

[5]Currently, a resolver bug between tuple types and references means tuple routines must use pointer parameters. Nevertheless, the imaginary reference versions are used here as the code is cleaner, *e.g.*, no & on call arguments.

Case	Loop headers	Outputs
(i) Iterating forward and reverse through the entire list. Throughout, note that the loop's middle condition "⁢" implies "&it != 0p."	**for** (node & it = first(list); ⁢ &it = &next(it)) ... **for** (node & it = last(list); ⁢ &it = &prev(it)) ...	1, 2, 3, 4, 4, 3, 2, 1,
(ii) Iterating forward and reverse from a starting node through the remaining list.	**for** (node & it = n2; ⁢ &it = &next(it)) ... **for** (node & it = n3; ⁢ &it = &prev(it)) ...	2, 3, 4, 3, 2, 1,
(iii) Iterating forward and reverse from a starting node to an ending node through the contained list.	**for** (node & it = n2; &it != &n4; &it = &next(it)) ... **for** (node & it = n4; &it != &n2; &it = &prev(it)) ...	2, 3, 4, 3,
(iv) Iterating forward and reverse through the entire list using the shorthand to start at the list head.	**for** (node & it = iter(list); advance(it); /*1*/) ... **while** (node & it = iter(list); advance(it)) ... **while** (node & it = iter(list); recede(it)) ...	1, 2, 3, 4, 1, 2, 3, 4, 4, 3, 2, 1,
(v) Convenience macros, iterating forward and reverse through the entire list.	FOREACH(list, it) ... FOREACH_REV(list, it) ...	1, 2, 3, 4, 4, 3, 2, 1,
(vi) Convenience macros, iterating forward and reverse through the entire list or until a predicate is triggered.	FOREACH_COND(list, it, it.v == 3) ... FOREACH_REV_COND(list, it, it.v == 1) ...	1, 2, 4, 3, 2,

Figure 4.7: Linked list iteration examples. Each loop header replaces the stub header of Figure 4.6.

Finally, the manual-iteration loops of (a–iv) also take predicates, as in (vi), but by simple composition:

for (node & it = first(list); &it && !(it.v == 3); &it = &next(it)) ...	1, 2,
for (node & it = last(list); &it && !(it.v == 1); &it = &prev(it)) ...	4, 3, 2,
for (node & it = iter(list); advance(it) && !(it.v == 3);) ...	1, 2,
for (node & it = iter(list); recede(it) && !(it.v == 1);) ...	4, 3, 2,

4.4 C++ Lists

It is worth addressing two API issues in C++ lists avoided in C∀. First, C++ lists require two steps to remove a node versus one in C∀.

C++

```
list<node> li;
node n = li.first();  // assignment could raise exception
li.pop_front();
```

C∀

```
dlist(node) list;
node n = remove_first( list );
```

The argument for two steps is exception safety: returning an unknown T by value/move might throw an exception from T's copy/move constructor. Hence, to be *exception safe*, all internal list operations must complete before the copy/move so the list is consistent should the return fail. This coding style can result in contrived code, but is usually possible; however, it requires the collection designer to anticipate the potential throw. (This anticipation issue is pervasive in exception systems, not just with collections.) The solution moves the coding complexity from the collection designer to the programer [52, ch 10, part 3]. First, obtain the node, which might fail, but

C++	C∀
list<node *> list;	dlist(node) list;
node n1 = { 1 }, n2 = { 2 }, n3 = { 3 }, n4 = { 4 };	node n1 = { 1 }, n2 = { 2 }, n3 = { 3 }, n4 = { 4 };
list.push_back(&n1); list.push_back(&n2);	insert(list, n1, n2, n3, n4);
list.push_back(&n3); list.push_back(&n4);	
list<node *>::iterator it;	
for (it = list.begin(); it != list.end(); it ++)	
if (*it == &n3) { dl.erase(it); **break**; }	remove(list, n3);

Figure 4.8: Obtaining a linked-list iterator in C++ *vs.* C∀

the collection is unmodified. Second, remove the node, which modifies the collection without the possibly of an exception. This movement of responsibility increases the cognitive effort for programmers. Unfortunately, this *single-responsibility principle*, *i.e.*, preferring separate operations, is often repeated as a necessary requirement rather an an optional one. Separate operations should always be available, but their composition should also be available. Interestingly, this issue does not apply to intrusive lists, because the node data is never copied/moved in or out of a list; only the link fields are accessed in list operations.

Second, Figure 4.8 shows an example where a C++ list operation is $O(N)$ rather than $O(1)$ in C∀. This issue is inherent to wrapped (non-intrusive) lists. Specifically, to remove a node requires access to the links that materialize its membership. In a wrapped list, there is no access from node to links, and for abstraction reasons, no direct pointers to wrapped nodes, so the links must be found indirectly by navigating the list. The C++ iterator is the abstraction to navigate wrapped links. So an iterator is needed, not because it offers go-next, but for managing the list membership. Note that attempting to keep an array of iterators to each node requires high-complexity to ensure the list and array are harmonized.

4.5 Implementation

Figure 4.9 continues the running req example, showing the C∀ list library's internal representation. The dlink structure contains exactly two pointers: next and prev, which are opaque. Even though the user-facing list model is linear, the C∀ library uses a circular model under the hood. This choice helps achieve uniform end treatment. A link pointer targets a neighbouring dlink structure, rather than a neighbouring req. (Recall that the running example has the user putting a dlink within a req.)

Circular link-pointers (dashed lines) are tagged internally in the pointer to indicate linear endpoints. Links among neighbour nodes are not tagged. Iteration reports "has more elements" when accessing untagged links, and "no more elements" when accessing tagged links. Hence, the tags are set on the links that a user cannot navigate.

The C∀ library works in headed and headless modes. In a headed list, the list head (dlist(req)) acts as an extra element in the implementation-level circularly-linked list. The content of a dlist header is a (private) dlink, with the next pointer to the first element, and the prev pointer to the last element. Since the head wraps a dlink, as does req, and since a link-pointer targets a dlink, the resulting cycle is among dlink structures, situated inside of header/node. An untagged pointer

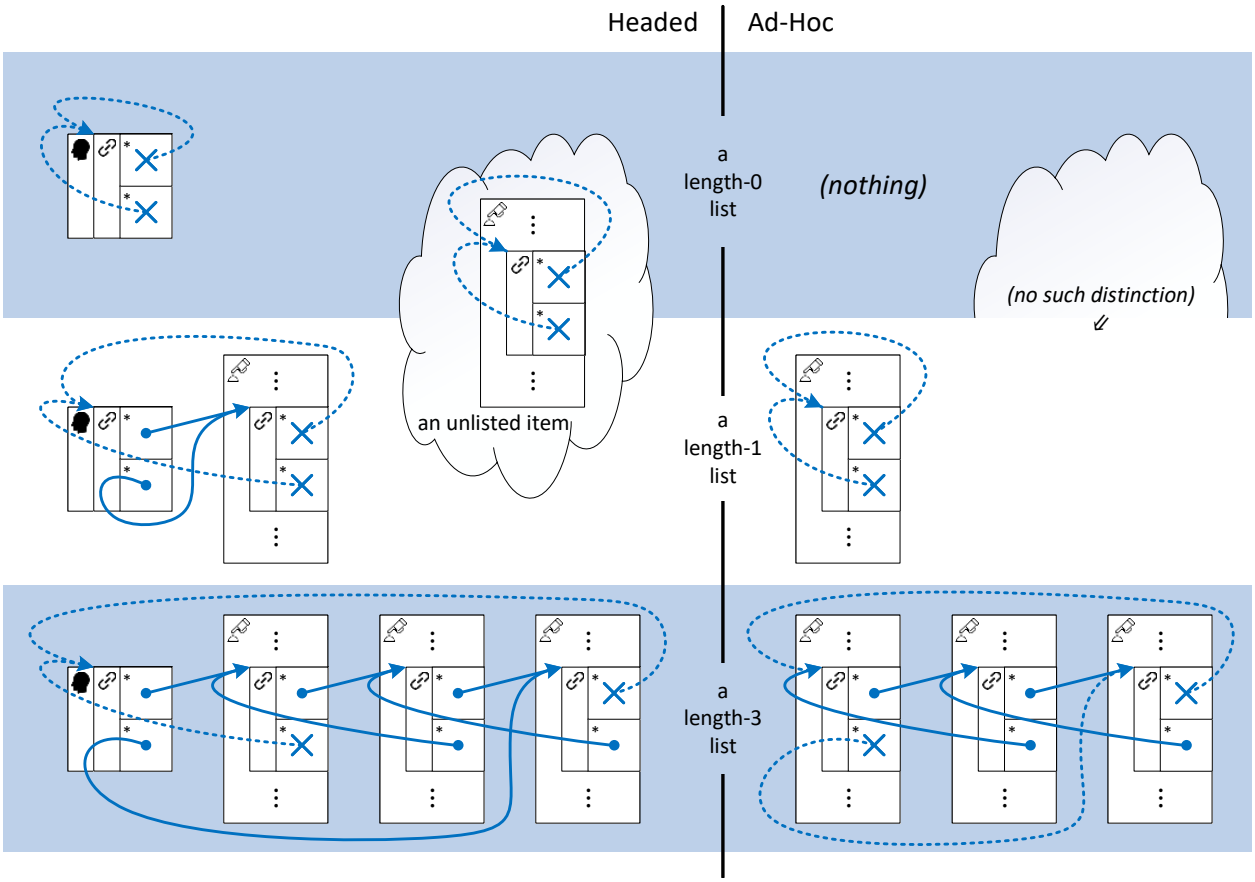


Figure 4.9: C∀ list library representations for headed and headless lists

points within a req, while a tagged pointer points within a list head. In a headless list, the circular backing list is only among dlinks within reqs.

No distinction is made between an unlisted node (top left and middle) under a headed model and a singleton list under a headless model. Both are represented as an item referring to itself, with both tags set.

4.6 Assessment

This section examines the performance of the discussed list implementations. The goal is to show the C∀ list is competitive with other designs, but the different list designs may not have equivalent functionality, so it is impossible to select a winner encompassing both functionality and execution performance.

Moreover, the performance comparisons, even considered apart from feature support, show wildly unpredictable responses to apparently minor differences in testing scenario. Execution times for intrusive linked-list operations are incredibly short, once coarser determinants like inherent cache pressure are removed. The linked-list implementations tend to show one coherent performance story under a given scenario, then a different coherent story in another scenario. A localized

inspection tempts with winner-loser hypotheses quickly, while a more global comparison suggests that there is no dominator.

4.6.1 Experiment Design

This section explains how the experiment is built. Many of the following parts define terminology concerning tuning knobs. Table 4.1 provides a consolidated reference.

4.6.1.1 Add-Remove Performance

The fundamental job of a linked-list library is to manage the links that connect nodes. Any link management is an action that causes pair(s) of elements to become, or cease to be, adjacent in the list. Thus, adding and removing an element are the sole primitive actions.

Repeated adding and removing is necessary to measure timing because these operations can be as short as a dozen instructions. These instruction sequences may have cases that proceed (in a modern, deep pipeline) without a stall.

This experiment takes the position that:

1. The total time to add and remove is relevant, as opposed to having one time for adding and a separate time for removing. Adds without removes quickly fill memory; removing without adding is impossible.[6]
2. A relevant breakdown "by operation" is, rather, the usage pattern of the add/remove calls. An example pattern choice is adding and removing at the same end, making a stack, or opposite ends, for a queue. Another is pushing on the front by calling insert_first(lst, e) *vs.* insert(e, e_ref), where e_ref is independently known to be the first element. This aspect provides the test's API coverage. Section 4.6.1.3, p. 93 gives the full breakdown.
3. Speed differences caused by the host machine's memory hierarchy need to be identified and explained, but do not represent advantages of one framework over another.

The experiment used to measure IR cost measures the mean duration of a sequence of additions and removals. The distribution of speeds experienced by an individual add-remove pair (tail latency) is not discussed. Space efficiency is shown only indirectly, by way of caches' impact on speed. The experiment is sensitive enough to show:

- intrusive lists performing (majorly) differently than wrapped lists,
- a space of (lesser) performance differences among the intrusive lists.

In all cases, the quantity discussed is the duration of one insert-remove (IR). An IR is the time taken to do one innermost insertion-loop iteration, one innermost removal-loop iteration, and its share of all overheads, amortized. Lower IR is better. This experiment typically does an IR in 1–10 ns. The short end of this range has durations of single-digit clock-cycle counts. Therefore, the situations that achieve the best times are saturating the instruction pipeline successfully.

Often, an IR duration value needs to be considered relatively. For example, Section 4.6.4, p. 101 asks whether one linked list implementation is more sensitive than another to the computer

[6] Section 4.7.2, p. 102 identifies situations that run counter to this intuition.

Table 4.1: Glossary of terms used in the list performance evaluation

Insert-Remove (IR)	The atomic unit of work being measured: one insertion plus one remove (plus all looping/tracking overheads)
Use Case	Pattern of add-remove calls.
– Movement	
∋ stack	IRs happen at the same end.
∋ queue	IRs happen at opposite ends.
– Polarity	Which of the two orientations in which the movement happens.
∋ insert-first	All inserts at front; stack removes at front; queue removes at back.
∋ insert-last	All inserts at back; stack removes at back; queue removes at front.
– Accessor	How an insertion position, or removal element, is specified. The same position/element is picked either way.
∋ all head	IRs both through the head
∋ insert element	insert by element and remove through the head
∋ remove element	insert through head and remove by element
Physical Context	
– Size (number)	Number of nodes being linked. Equals the length of the program's sole list.
– Size Zone	Contiguous range of sizes, chosen to avoid known anomalies and to sample a brief plateau. Each zone buckets four specific sizes.
∋ small	lists of 4–16 elements
∋ medium	lists of 50–200 elements
∋ (other)	Not used for comparing intrusive frameworks.
– Machine	Computer running the experiment
∋ AMD	smaller cache
∋ Intel	bigger cache
Framework	A particular linked-list implementation (within its host language)
∋ C++	The std::list type of g++.
∋ lq-list	The list type of LQ from glibc of gcc.
∋ lq-tailq	The tailq type of the same.
∋ μC++	μC++'s uSequence
∋ C∀	C∀'s dlist
Individual config'n (IC)	Specific situation in which an IR sequence is timed. One use case, on one machine, driving one exact list size, by one framework.
Trial	Timed run of the test program under a given IC, during which a few billion IRs occur.
Explanation being	How independent explanatory variable X is analyzed
– Marginalized	Left alone, allowed to vary, yielding a more absolute measure. Shows the effect that X causes. If all explanations are marginalized, then absolute times are available and a relative time has a peer group that is the entire population.
– Conditioned	Held constant, yielding a more relative measure. Hides the effect that X causes. Conditioning on X creates more, smaller relative-measure peer groups, by isolating each X-domain value. Resulting interpretation is, "Assuming no change in X."

architecture. A finding might be that a machine slows implementation A by 10% and B by 20%. This finding is not saying that A is faster than B (on either machine). The finding could stand if B starts faster and then levels off, if B starts slower and gets worse, or in myriad other cases. The finding asserts that such distinctions are not what is immediately relevant. The arithmetic producing different answers is removing the information about which one starts or ends up faster. Each implementation's to-machine duration is stated relatively to *the same implementation's* from-machine duration. The resulting measure is still about a duration. The framework with the lower from-machine-relative duration handles the change better.

4.6.1.2 Test Program

The experiment driver defines a (intrusive) node type:

```
struct Node {
    int i, j, k;  // fields
    // possible intrusive links
};
```

and considers the speed of building and tearing down a list of n instances of it.

```
// simplified harness: CFA implementation,
// stack movement, insert−first polarity, head−mediated access
size_t totalOpsDone = 0;
dlist( node_t ) lst;
node_t nodes[ n ];                          // preallocated list storage
startTimer();
while ( CONTINUE ) {                        // ≈ 20 second duration
    for ( i; n ) insert_first( lst, nodes[i] );   // build up
    for ( i; n ) remove_first( lst );             // tear down
    totalOpsDone += n;
}
stopTimer();
reportedDuration = getTimerDuration() / totalOpsDone; // throughput per IR operation
```

To reduce administrative overhead, the n nodes for each experiment list are preallocated in an array (on the stack), which removes dynamic allocations for this storage. These nodes contain an intrusive pointer for intrusive lists, or a pointer to it is stored in a dynamically-allocated internal-node for a wrapper list. Copying the node for wrapped lists skews the results with administration costs. The list insertion/removal operations are repeated for a typical 20+ second duration. After each round, a counter is incremented by n (for throughput). Time is measured outside the loop because a large n can overrun the time duration before the CONTINUE flag is tested. Hence, there is a minimum of one outer (CONTINUE) loop iteration for large lists. The loop duration is divided by the counter and this throughput is reported. In a scatter-plot, each dot is one throughput, which means insert + remove + harness overhead. The harness overhead is constant when comparing linked-list frameworks and is kept as small as possible.

To test list operations, the experiment performs the inserts/removes in different patterns, *e.g.*, insert and remove from front, insert from front and remove from back, random insert and remove, *etc*. Unfortunately, the std::list does *not* support direct IR from a node without an iterator, *i.e.*, no erase(node), even though the list is doubly-linked. In use cases like *remove-here*, keeping

these iterators is a legitimate cost of the use case. Iterators, which are trivially element pointers for intrusive lists, live in a separate array, which implies its own stream of memory accesses and consequent cache impact.

Use cases that do not depend on such extra access streams would not be measured accurately if the harness imposed its own. So, a side array used for random indirection, nodes[shuf[i]], must be rejected. A trick is used for random insertions, which takes advantage of the array allocation of the nodes. The i fields in each node are initialized from 0..n−1. These i values are then shuffled in the nodes, and the i value is used to represent an indirection to the node for insertion. The shuffling applies a single-cycle permutation. Hence, every node is inserted, once, in random order.

```
for ( i; n ) nodes[i].i = i;                    // indirection
shuffle( nodes, n );                            // random shuffle indirects within nodes

while ( CONTINUE ) {
    node * cur = nodes[0];
    for ( i; n ) {
        insert_first( lst, *cur );              // build up
        cur = & nodes[ cur->i ];                // select next random node in array
    }
    for ( i; n ) pass( &remove_first( lst ) );  // tear down
    totalOpsDone += n;
}
```

Removal follows the same random order, by virtue of following the built-up links.

Insertion is traversing the list in random order. For intrusive lists, the inserted (random) node is always touched because its link fields are modified for insertion into the list. For wrapped lists, the wrapped nodes are still being treaded in random order, but only a pointer to a node is inserted into a linearly accessed wrapper. The indirection mechanism provides an equalizer, so that the wrapped experiment is also accessing each node, while it does an IR. Furthermore, it is rare to IR nodes and not access them.

4.6.1.3 Use Cases

Where Table 4.1, p. 91 enumerates the specific values, recall the use-case dimensions are:

movement ($\times 2$) In these experiments, strict stack and queue patterns are tested.

polarity ($\times 2$) Obtain one polarity from the other by reversing uses of first/last.

accessor ($\times 3$) Giving an add/remove location by a list head's first/last, *vs.* by a preexisting reference to an individual element.

A use case is a specific selection of movement, polarity and accessor. These experiments run twelve use cases. When a comparison is showing only what can happen when switching among use cases (as opposed to *e.g.*, how stacks are different from queues), the numbering scheme of Figure 4.10 is used.

	I	II	III	IV	V	VI	VII	VIII	IX	X	XI	XII
Movement	stack	stack	stack	stack	stack	stack	queue	queue	queue	queue	queue	queue
Polarity	i-first	i-first	i-first	i-last	i-last	i-last	i-first	i-first	i-first	i-last	i-last	i-last
Accessor	all hd	ins-e	rem-e	all hd	ins-e	rem-e	all hd	ins-e	rem-e	all hd	ins-e	rem-e

Figure 4.10: Experiment use cases, numbered

With accessor, when an action names its insertion position or removal element, the harness either

- defers to the list-head's tracking of first/last ("through the head"), or
- applies its own knowledge of the current pattern, to name a position/element that happens to be first/last ("of known element").

The accessor patterns, at the (C∀) API level, are:

all (through the) head: Both IRs happen through the list head. The list head operations are insert_first, insert_last, remove_first and remove_last.

insert (of known) element ... and remove through head: Inserts use insert_before(e, first) or insert_after(e, last), where e is being inserted and first/last are element references known by list-independent means.

remove (of known) element ... and insert through the head: Removes use remove(e), where e is being removed. List-independent knowledge establishes that e is first or last, as appropriate.

Comparing all-head with insert-element gives the relative performance of head-mediated *vs.* element-oriented insertion, because both use the same removal style. Comparing all-head with remove-element gives the relative performance of head-mediated *vs.* element-oriented removal, because both use the same insertion style.

4.6.1.4 Sizing

Intuition suggests measuring IR for different sized lists should just be a multiple of the single linking/unlinking of a node. However, there is a scaling issue as more memory is being read and written, where caching comes into play. But caching is more than the amount of memory being accessed; the access pattern is equally important. Aggressive instruction-level parallelism scheduling, which enables short IR times, is the amplifier, *e.g.*, a data dependency is a critical path in one situation but not in another. Therefore, the duration's response to size is not a steady worsening as size increases. Often, each size-independent configuration responds to size increases in steps of slowdown. Occasionally a slowdown step is followed by some performance increase, where an incurred penalty begins to amortize away. Hence, performance results can have interesting jitter as size increases. The analysis treats these behaviours as incidental. It does not try to characterize various exact-size responses. Rather, size zones are picked, specific effects inside of a zone are averaged away, and the story at one zone is compared to that at another.

Figure 4.11 gives two example responses to size. These two example cases show how differently a pair of configurations can behave. Outside of an identified box, size response is erratic. Inside a box, size response is relatively smooth. Within and among the boxes there are identifiable patterns, which occur throughout all the experimental results.

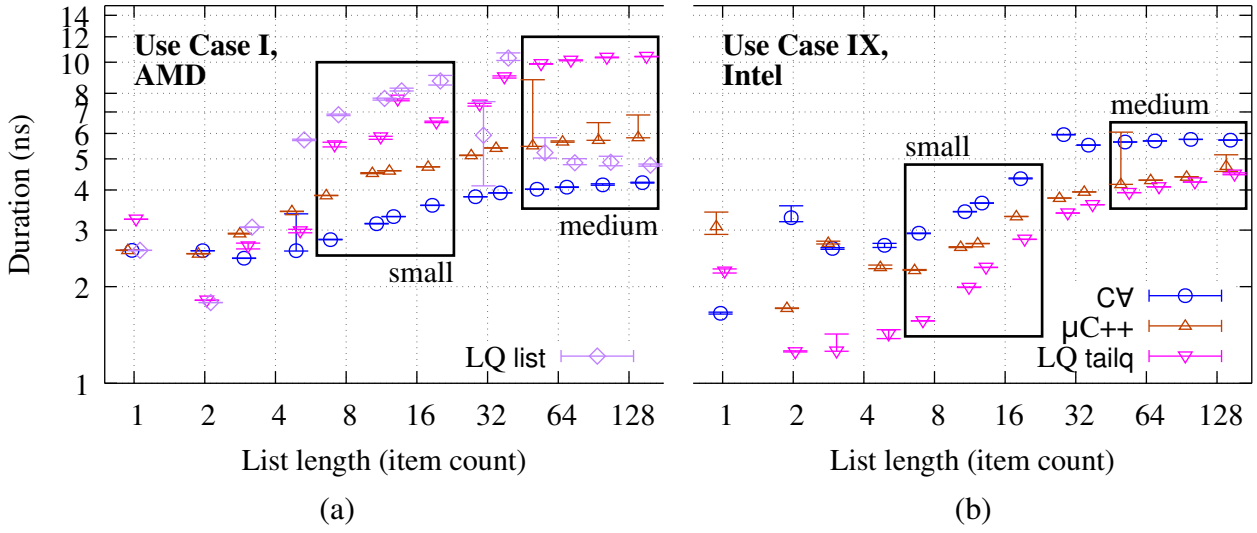


Figure 4.11: Variety of IR duration responses to list length, at small–medium lengths. Two example use cases are shown: I, stack movement with head-only access (plot a); IX, queue movement with element-oriented removal access (plot b); both use cases have insert-first polarity. One example is run on each machine: UC-I on AMD (plot a); UC-IX on Intel (plot b). Lower is better.

Every data point is one Individual Configuration (IC). Each IC is tested by five trials; a point's error bars give the best and worst trial results; the point's centre is the mean of the middle three. The amount of error here is typical across all the configurations (beyond the two shown here). With a few exceptions, it is modest, so these experiments are repeatable.

To preview, Section 4.6.2, p. 97 dismisses large sizes (above 150 elements) and wrapped lists, because the performance story is dominated by the amount of memory touched, not by intrusive *vs.* wrapped lists. At smaller sizes, Section 4.6.3, p. 99 shows differences appear among the intrusive-list implementations. Among the "not Large" sizes ($\leq$ 150), two zones, Small and Medium, are selected as representatives of what can vary when the scale is changed. These particular ranges are chosen because each range tends to have one story repeated across its constituent sizes. For example, if C∀'s duration increases across Small, then the other frameworks' usually do too, or if C∀ is beating μC++ across Medium, then it usually is at the high end too.

Finally, on the AMD architecture, C∀ performed poorly at size 1, on queue movements only, and no other framework saw the same effect. This extreme outlier is not plotted in graphs. After exploring the phenomenon in depth (not presented), the conclusion is a quirky interaction between the hardware and the testing harness. A side experiment (that does not enrich the overall comparisons) saw user-induced gaps of $\approx$10 ns between same-list operations hide the effect completely. These gaps are realistic because when an item goes on a list another action comes back to it *later*. The pattern that the general harness uses, concentrating time-adjacent operations on one list, is useful for measuring the "small" size zone, but is contrived, from the perspective of a data hazard that only this pattern exposes. The general comparisons do not see the effect at all, because they use only the Small and Medium zones, with shortest length of 4.

4.6.1.5 Execution Environment

The performance experiments are run on:

AMD Supermicro AS–1125HS–TNR EPYC 9754 128–core socket, hyper-threading × 2 sockets (512 processing units) 2.25 GHz, TSO memory model, with cache structure 32KB L1i/L1d, 1024KB L2, 16MB L3, where each L3 cache covers 1 NUMA node and 8 cores (16 processors).

Intel Supermicro SYS-121H-TNR Xeon Gold 6530 32–core, hyper-threading × 2 sockets (128 processing units) 2.1 GHz, TSO memory model, with cache structure 32KB L1i/L1d, 20248KB L2, 160MB L3, where each L3 cache covers 2 NUMA node and 32 cores (64 processors).

An experiment run is single-threaded. To prevent perturbation from cache or NUMA effects, the thread is pinned to a single core to prevent any OS movement.

The compiler is gcc/g++-14.2.0 running on the Linux v6.8.0-52-generic OS. To prevent eliding certain code patterns, crucial parts of a test are wrapped by the function pass

```
// prevent eliding, cheaper than volatile
static inline void * pass( void * v ) { __asm__ __volatile__( "" : "+r"(v) ); return v; }
...
pass( &remove_first( lst ) );          // wrap call to prevent elision, insert cannot be elided now
```

The call to pass can prevent a small number of compiler optimizations but this cost is the same for all lists.

C∀ runs the llheap allocator [58], which is significantly better than the standard glibc allocator. To obtain fair comparisons of allocator impact, this experiment plugs llheap into C++. Switching between the default memory allocators glibc and llheap is done with LD_PRELOAD.

The main difference in the machines is their cache structure. The AMD has smaller caches that are shared less, while the Intel shares larger caches among more processors. This difference, while an interesting tradeoff for highly concurrent use, is rather one-sided for sequential use, such as this experiment. Specifically, the Intel offers a single processor a bigger cache.

4.6.1.6 Recap and Master Legend

For experiments performed in later sections, there are 12 use cases, which are all combinations of 2 movements, 2 polarities and 3 accessors. There are 4 physical contexts, which are all combinations of 2 machines and 2 size (length) zones. There are 3.25 frameworks. This accounting considers how LQ-list supports only the movement–polarity combination "stack, insert first." So LQ-list fills a quarter of the otherwise-orthogonal space. Physical context, use case, and framework are the explanatory factors. Each size zone summarizes samples of 4 specific sizes. Taking all combinations of the explanatory factors and this sampling gives $4 \times 12 \times 3.25 \times 4 = 624$ individual configurations (ICs).

A condensed graphing style is used in subsequent plots to present this amount of data. Figure 4.12 shows how the condensed graphing style is generated from individual-configuration measures. Figure 4.12(a) is formed from the data in Figure 4.11(a), transformed on the Y-axis to show duration relative to the mean across all four frameworks, at each specific size. Figure 4.12(b) condenses the interesting data within the two boxes (Small/Medium) and their combination (Both).

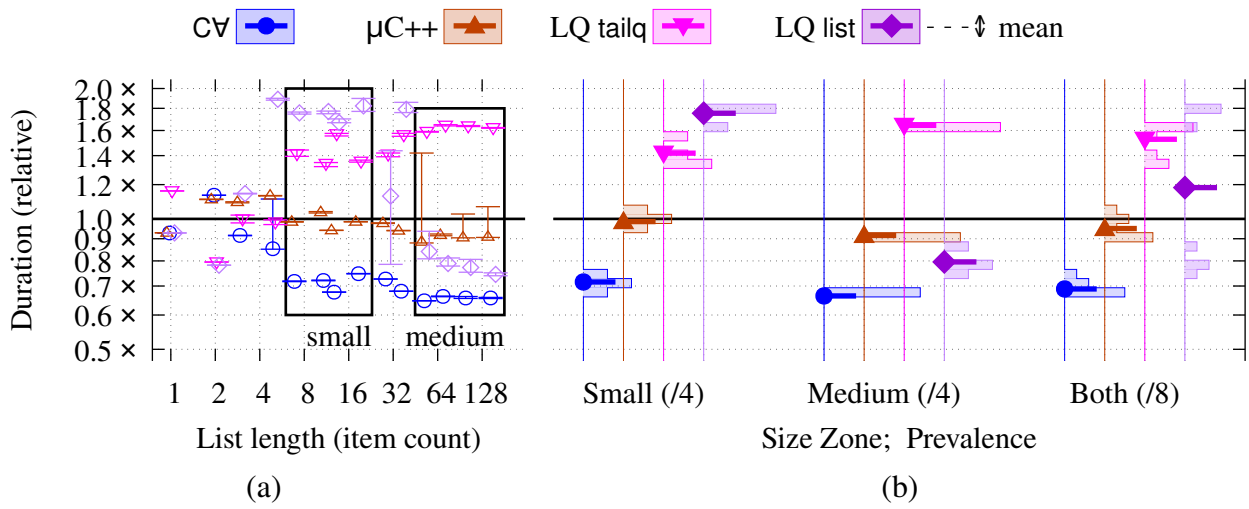


Figure 4.12: IR duration, transformed for general analysis. The analysis follows the single example setup of Figure 4.11(a), p. 95, *i.e.*, Use Case I on AMD; there, IR is given as absolute duration. Plot (a) transforms the source dataset by conditioning on specific size. Plot (b) takes the results from only the identified size zones, discards their specific-size information, and shows the resulting distribution. Lower is better.

This graph plots a vertical histogram for each of the 4 frameworks. A data point on Figure 4.11(a), p. 95 is one-to-one with a point on Figure 4.12(a); each gives one IC. Repeatability of the experiment being established previously, retry variance information (error bar on an IC point) is discarded, and only the expected performance of an IC (mean of its middle three out of five trials) is promoted into the histograms. Each histogram bin (light-shaded area) counts the number of ICs whose expected performance falls in the bin's range. A histogram's girth indicates the diversity of its qualifying configurations' performance expectations. The overlaid tick symbol marks its group's mean performance. An x label indicates the total number of ICs in a group, *e.g.*, "/4" $\Rightarrow$ 4 ICs per histogram (here, a size-zone/framework combination). The totals are small here because attention is still restricted to the example slice of Use Case I on AMD. But the graph format now scales to handle the full population of tested configurations, which is the subject of all subsequent plots.

4.6.2 Result: Coarse Comparison of Styles

This comparison establishes how an intrusive list performs compared with a wrapped-reference list. Figure 4.13 presents throughput at various list lengths for a linear and random (shuffled) IR test. Other kinds of scans were made, but the results are similar in many cases, so it is sufficient to discuss these two scans, representing difference ends of the access spectrum. In the graphs, all four intrusive lists (lq-list, lq-tailq, μC++, C∀, see Framework in Table 4.1, p. 91) are plotted with the same symbol; sometimes theses symbols clump on top of each other, showing the performance difference among intrusive lists is small in comparison to the wrapped list (std::list). See Section 4.6.3, p. 99 for details among intrusive lists.

The list lengths start at 10 due to the short IR times of 2–4 ns for intrusive lists *vs.* STL's wrapped-reference list of 15–20 ns. For very short lists, like 4, the experiment time of 4 $\times$ 2.5 ns

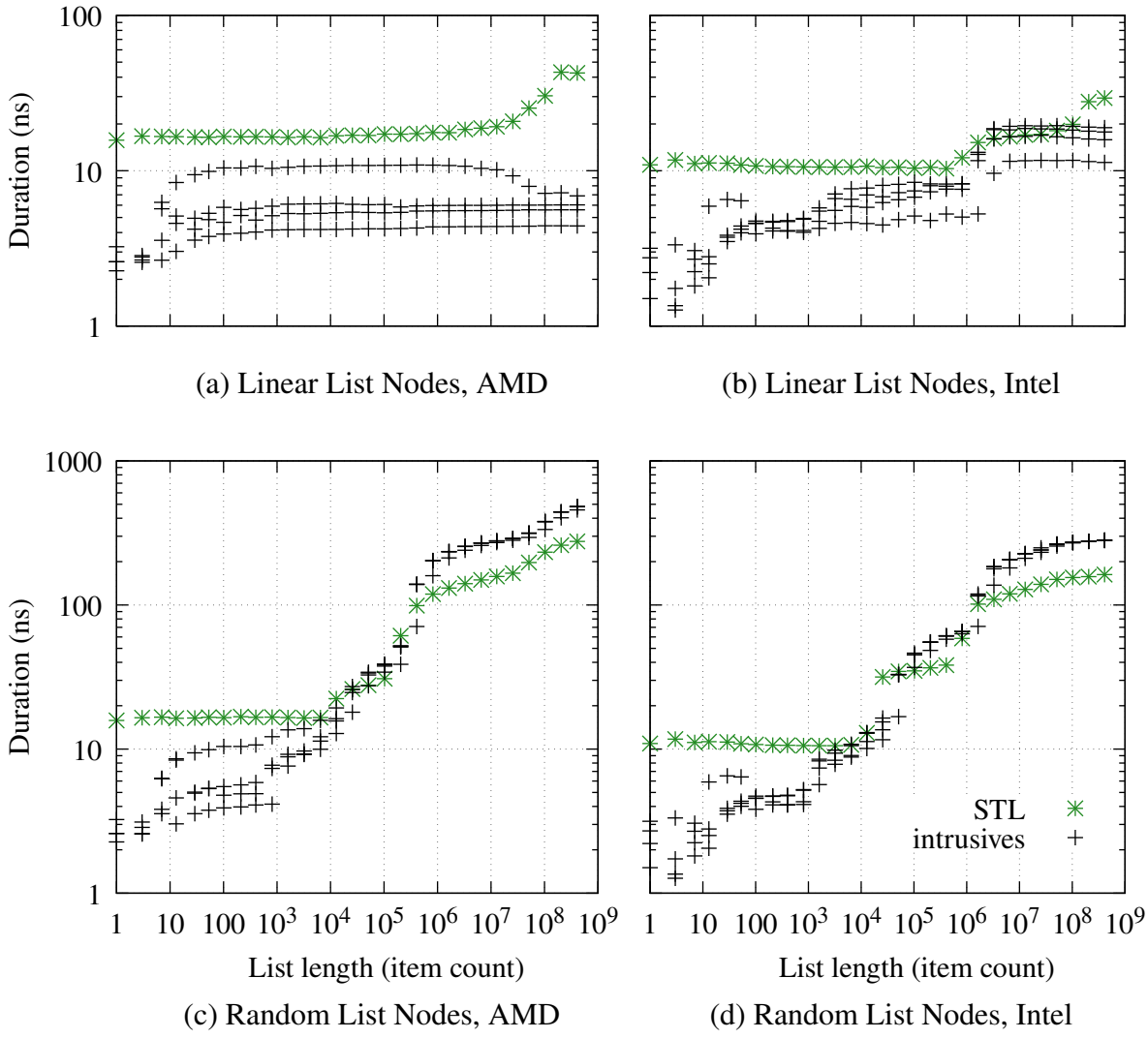


Figure 4.13: IR duration *vs.* list length, all sizes. Lengths go as large possible without error. One example use case is shown: stack movement, insert-first polarity and head-mediated access. Lower is better.

and experiment overhead (loops) of 2–4 ns, results in an artificial administrative bump at the start of the graph having nothing to do with the IR times. As the list size grows, the administrative overhead for intrusive lists quickly disappears.

The key performance factor between intrusive and wrapped-reference lists is the dynamic allocation for the wrapped nodes. Hence, this experiment is largely measuring the cost of malloc/free rather than IR, and is sensitive to the layout of memory by the allocator. For intrusive-list IR, the cost is manipulating the link fields, which is seen by the relatively similar results for the different intrusive lists. For wrapped-reference IR, the costs are: dynamically allocating/deallocating a wrapped node, copying a external-node pointer into the wrapped node for insertion, and linking the wrapped node to/from the list; the allocation dominates these costs. For example, the experiment was run with both glibc and llheap memory allocators, where llheap performance reduced the cost

from 20 to 16 ns, still far from the 2–4 ns for linking an intrusive node. Hence, there is no way to tease apart the allocation, copying, and linking costs for wrapped lists, as there is no way to preallocate the list nodes without writing a mini-allocator to manage that storage.

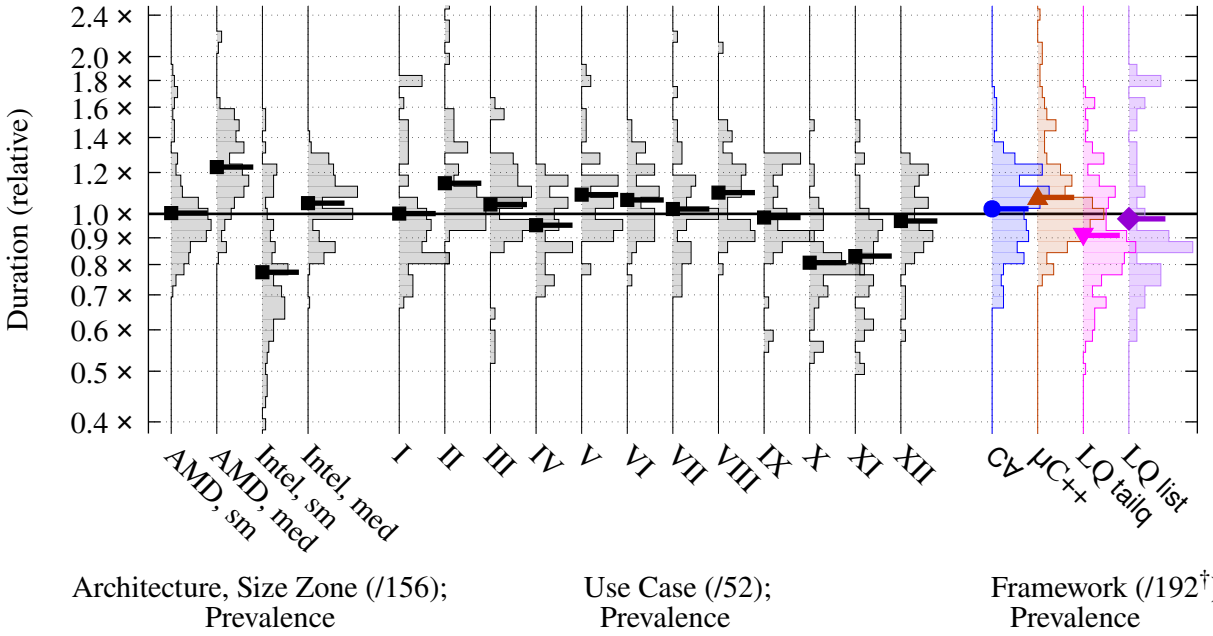
In detail, Figure 4.13(a)–(b) shows linear insertion of all the nodes and then linear removal, both in the same direction. For intrusive lists, the nodes are adjacent in memory from being preallocated in an array. For wrapped lists, the wrapped nodes happen to be adjacent because the memory allocator uses bump allocation during the initial phase of allocation. As a result, these memory layouts result in high spatial and temporal locality for both kinds of lists during the linear array traversal. With address look-ahead, the hardware does an excellent job of managing the multi-level cache. Hence, performance is largely constant for both kinds of lists, until L3 cache and NUMA boundaries are crossed for longer lists and the costs increase consistently for both kinds of lists. For example, on AMD (Figure 4.13(a)), there is one NUMA node but many small L3 caches, so performance slows down quickly as multiple L3 caches come into play, and remains constant at that level, except for some anomalies for very large lists. On Intel (Figure 4.13(b)), there are four NUMA nodes and four slowdown steps as list-length increase. At each step, the difference between the kinds of lists decreases as the NUMA effect increases.

In detail, Figure 4.13(c)–(d) shows random insertion and removal of the nodes. As for linear, there is the issue of memory allocation for the wrapped list. As well, the consecutive storage-layout is the same (array and bump allocation). Hence, the difference is the random linking among nodes, resulting in random accesses, even though the list is traversed linearly, resulting in similar cache events for both kinds of lists. Both Figures 4.13(c)–(d) show the slowdown of random access as the list-length grows resulting from stepping out of caches into main memory and crossing NUMA nodes. As for linear, the Intel (Figure 4.13(d)) graph shows steps from the four NUMA nodes. Interestingly, after 10^6 nodes, intrusive lists are slower than wrapped. I did not have time to track down this anomaly, but I speculate it results from the difference in touching the data in the accessed node, as the data and links are together for intrusive and separated for wrapped. For the llheap memory-allocator and the two tested architectures, intrusive lists out perform wrapped lists up to size 10^3 for both linear and random, and performance begins to converge around 10^6 nodes as architectural issues begin to dominate. Clearly, memory allocator and hardware architecture plays a large factor in the total cost and the crossover points as list-size increases.

The takeaway from this experiment is that wrapped-list operations are expensive because memory allocation is expense at this fine-grained level of execution. Hence, when possible, using intrusive links can produce a significant performance gain, even if nodes must be dynamically allocated, because the wrapping allocations are eliminated. Even when space is a consideration, intrusive links may not use more storage if a node is often linked. Unfortunately, many programmers are unaware of intrusive lists for dynamically-sized data-structures or their tool-set does not provide them.

4.6.3 Result: Intrusive Winners and Losers

The preceding result shows the intrusive frameworks have better performance than the wrapped lists for small to medium sized lists. This analysis covers the experiment position taken in Section 4.6.1.1, p. 90 for movement, polarity, and accessor. Figure 4.10, p. 94 shows the experiment



† LQ-list is (/48) by its incomplete (25%) use case coverage. Its bars are scaled to match.

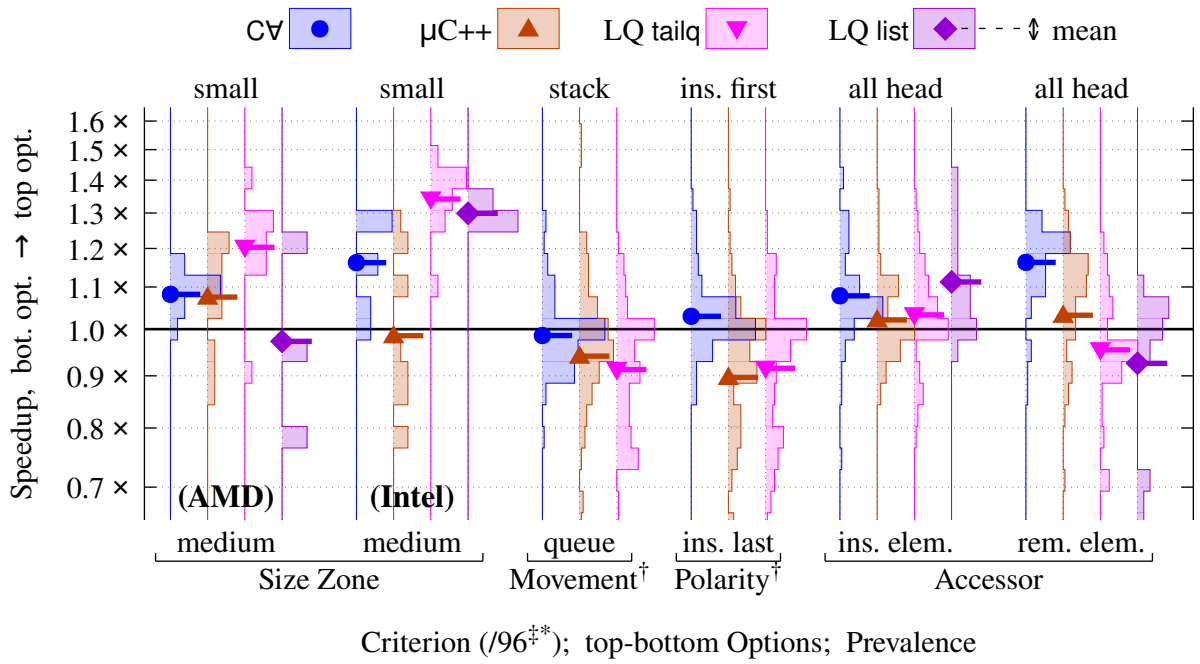
Figure 4.14: IR duration, decomposed by all first-order effects. Each of the three breakdowns divides the entire population of test results into its mutually disjoint constituents. Lower is better.

use cases tested, which results in 12 experiments (I–XII) for comparing intrusive frameworks. To preclude hardware interference, only list sizes below 150 are examined to differentiate among the intrusive frameworks, The data is selected from the start of Figures 4.13(a), p. 98–(b), but the start of Figures 4.13(c), p. 98–(d) is largely the same.

Figure 4.14 gives the first-order effects. The first breakdown, architecture/size-zone (left), shows the overall performance of all configurations, split by the two different hardware architectures and by small *vs.* medium lists (624 / 4 = 156 experiments per column). By inspection of the averages, Intel runs faster than AMD. Within an architecture, the small zone (lists of 4–16 elements) runs faster than the medium zone (lists of 50–200 elements). The overall slower execution on the AMD results from its smaller cache *vs.* the larger cache on the Intel. (No NUMA effects for these list sizes.) Specifically, a 20% standard deviation exists here, between the means of the four physical-effect categories. These hardware effects are accounted for when interpreting the following framework comparisons.

The second breakdown, use case (middle), shows the overall performance for each of the 12 use cases from Figure 4.10, p. 94 (624 / 12 = 52 experiments per column). The standard deviation of the individual use cases' means is 10%. A more detailed analysis occurs in the discussion of Figure 4.15.

The third breakdown, framework (right), shows the overall performance of the 4 list implementations (624 / 3.25 = 192). Here, C∀ runs similarly to μC++ and LQ-list runs similarly to tailq. The standard deviation of the frameworks' means is 7%. Now, C∀/μC++ run slower than LQ-list/tailq by



† LQ-list is absent from Movement and Polarity comparisons because it does not support queue and insert-last, respectively.

‡ LQ-list is (/24) by its incomplete (25%) use case coverage. Its bars are scaled to match.

* The full population of 192 individual configurations applies (48 for LQ-list), but this analysis summarizes pairs of them, giving each histogram's 96 contributions (24 for LQ-list).

Figure 4.15: IR duration where framework selection interacts with other factors. Higher favours top option; lower favours bottom option.

15%, a fact explored further in Section 4.6.4. But so too does use case X typically beat use case II by 38%. As does a small size on the Intel typically beat a medium size on the AMD by 66%. Hence, architecture and usage pattern have a more significant effect on speed than the selection of a framework.

4.6.4 Result: Sweet and Sore Spots

Figure 4.15 shows how frameworks react to a single other factor being varied across one pair of options. Every (binned and mean-contributing) individual data point represents a pair of test setups, one with the criterion set to the option labelled at the top; the other setup uses the bottom option. This point's y-axis score is the ratio of these setups' durations. The point lands in a bin closer to the label of the option that performs better.

The first breakdown, size zone (left), refines the notion that a small size runs faster than a big size; this issue is by how much. Indeed, all means favour small and few tails favour medium. But the various frameworks do no respond to the different sizes and machines uniformly. On the AMD, C∀ and μC++ have a modest size sensitivity, LQ-tailq's is moderate, and LQ-list seems unaffected. On the Intel, C∀'s increases to moderate, while μC++ is now unaffected, and both LQs have a large

effect. Hence, the Intel is more sensitive to size than the AMD.

The second breakdown, movement and polarity (middle), shows more-subdued responses. LQ-list does not show up in these comparisons because it only supports stacks that push and pop with the first element. C∀ is completely stable under movement and polarity changes. μC++ and LQ show modest responses favouring queues and insertion at last.

The third breakdown, accessor (right), shows close responses, except for C∀. Note that the three experiment setups are analyzed as a pair of two-way comparisons. First, the all-head/insert-element addresses which insertion style is better—by-head (top) and by-element (bottom). Then, the all-head/remove-element addresses which removal style is better—by-head (top) and by-element (bottom). The LQs favour insertion by head and removal by element. C∀ and μC++ favour both operations by head. The strongest effect is C∀'s aversion to removal by element—certainly an opportunity for improvement.

4.7 Future Work

4.7.1 Bounded Polymorphism

Not discussed in the chapter is a C∀ type-system issue with Plan-9 **inline** declarations, implementing the trait embedded to access the contained dlist link-fields. This trait defines an implicit conversion from derived to base (the safe direction). Such a conversion exists implicitly for concrete types using the Plan-9 inheritance feature. However, this implicit conversion is only partially implemented for polymorphism types, such as dlist, which prevents the straightforward list interface shown throughout the chapter.

My workaround is the macro P9_EMBEDDED placed before an intrusive node is used to declare a list.

```
struct node {
    int v;
    inline dlink(node);
};
P9_EMBEDDED( node, dlink(node) );
dlist( node ) dlist;
```

The macro creates specialized access functions to explicitly extract the required information for the polymorphic dlist type. These access functions could have been generated implicitly for each intrusive node by adding another compiler pass. However, this would have been substantial temporary work, when the correct solution is the compiler fix. Hence, the macro workaround is only a small temporary inconvenience; otherwise, all the list API shown in this chapter works.

4.7.2 Performance Assessment

The assessment of Section 4.6, p. 89 deals with the combined time for an addition, plus a corresponding removal. Isolating performance effects for additions *vs.* removals could also be valuable for cases where

- only one of the two operations is on the hot path, or
- the list exists within a context like region-based memory management, which obviates removing entirely.

Further searching for performance pitfalls could explore these corners.

The presented assessment is a suite of microbenchmarks. The usual limitation applies: these results could fail to generalize to real code. A next step in providing such an assurance would be to take some existing standard benchmark program that happens to use lists and port it to use C∀ and its list. This exercise would provide valuable learning, not only with its performance result, but also simply in identifying compatibility issues; not only in the linked-list realm, but in C∀ generally. Because C∀ has not yet undertaken such a port, this step could be a significant undertaking.

Chapter 5

String

This chapter presents my work on designing and building a modern string type in C∀. The discussion starts with an overview of the string API, then a number of interesting string problems, followed by how these issues are resolved in this work.

The need for a modern string type in C∀ might be questioned; C is far from a language of choice for working with text. Nevertheless, an adequate text-handling mechanism helps programmers get their jobs done. Firstly, C is avoided because of the difficulties outlined in Section 2.5, p. 33 and even headlined in the Chapter-1 infamous C issues. C∀ tries to treat such difficulties as fixable; so, this work attempts to elevate text processing from the anathema of the C world, up to the level of "no big deal." Secondly, nearly all programs include an element of string processing. It might be simply for command-line or configuration handling. To discover CPU-cache topology on Linux means reading through formatted text files under /sys/devices/system/cpu. But even these purposes introduce opportunities for programming mistakes, which can, in turn, lead to a compromised system. Finally, even within a naturally C-fitting program's core line of business, text processing creeps in. Consider a database; its text storage and indexing will no doubt necessitate a bespoke representation (as for its other datatypes); here, the ability to control individual bytes is precisely the reason for working in C. But the text does not simply sit stored; user requests imply processing upon the stored values: select first_name + last_name from.... These operations can produce and analyze intermediate text values, yet not amount to a significant reason for involving another programming language. In this last category, performance matters. Furthermore, the ubiquitous C++ idiom of passing text as **const** string & shows that avoiding unnecessary operations when handling text is an important consideration in C-realm programming. Therefore, this thesis seeks the joint goals of safety and performance from a general-purpose, data-in-motion string type in C∀.

5.1 String Operations

Figure 5.1 shows a general comparison of string APIs for C, C++, Java and C∀. It provides a classic "cheat sheet," summarizing the names of the most common closely-equivalent operations. The over-arching commonality is that operations work on groups of characters for assigning, copying, scanning, and updating.

C **char** [ ]	C++ string	Java String	C∀ string
strcpy, strncpy	=	=	=
strcat, strncat	+, +=	+, +=	+, +=
strcmp, strncmp	==, !=, <, <=, >, >=	equals, compareTo	==, !=, <, <=, >, >=
strlen	length, size	length	size
[ ]	[ ]	charAt	[ ]
strncpy	substr	substring	( ), on RHS of =
strncpy	replace	replace	( ), on LHS of =
strstr	find	indexOf	find
strcspn	find_first_of	matches	exclude
strspn	find_first_not_of	matches	include
N/A	c_str, data	N/A	strcpy, strncpy

Figure 5.1: Inter-language comparison of string APIs

As mentioned in Section 2.5, p. 33, a C string uses null termination rather than a length, which leads to explicit storage management; hence, most of its group operations are error prone and expensive due to copying. Most high-level string libraries use a separate length field and specialized storage management to implement group operations. Interestingly, C++ strings retain null termination in case it is needed to interface with C library functions, for example:

```
int open( const char * pathname, int flags );
string fname{ "test.cc" );
open( fname.c_str(), O_RDONLY );      // null terminated value of string
```

Here, the C++ c_str member does not create a new null-terminated C string from the C++ string, as that requires passing ownership of the C string to the caller for eventual deletion.[1] Providing this backwards compatibility with C has a ubiquitous performance and storage cost.

5.2 C∀ string Type

The C∀ string type is for manipulation of dynamically-sized character-strings versus C **char** * type for manipulation of statically-sized null-terminated character-strings. Therefore, the amount of storage for a C∀ string changes dynamically at runtime to fit the string size, whereas the amount of storage for a C string is fixed at compile time. As a result, a string declaration does not specify a maximum length, where a C string array does. For C∀, as a string dynamically grows and shrinks in size, so does its underlying storage. For C, as a string dynamically grows and shrinks in size, its underlying storage does not. The maximum storage for a C∀ string value is size_t bytes, which is 2^{32} or 2^{64} respectively.

In its present state, the C∀ string has partial support for Unicode. No overt support is included, though considerable Unicode support is implied. Sections 5.5, p. 141 and 5.6, p. 144 explain. For understanding the core features and implementation choices, this presentation assumes an ASCII or "string of bytes" mentality, to keep the Unicode discussion contained.

[1] C functions like strdup do return allocated storage that must be freed by the caller.

A C∀ string manages its length separately from the text, so there is no null (`'\0'`) terminating value at the end of a string value. Hence, a C∀ string cannot be passed to a C string manipulation function, such as strcat. Like C strings, characters in a string are numbered from the left starting at 0 (because subscripting is zero-origin), and in C∀ numbered from the right starting at -1.

```
"a      b      c      d      e"
 0      1      2      3      4       left to right index
-5     -4     -3     -2     -1       right to left index
```

The following operations manipulate an instance of type string, where the discussion assumes the following declarations.

```
#include <string.hfa>
string s = "abcde", name = "MIKE", digit = "0123456789";
const char cs[] = "abc";
int i;
```

Note the include file <string.hfa> to access type string.

5.2.1 Implicit String Conversions

The ability to convert from internal (machine) to external (human) format is useful in situations other than I/O. Hence, the basic types **char**, **char** *, **int**, **double**, **_Complex**, including any signness and size variations, implicitly convert to type string (as in Java).

```
string s = 5;                    "5"      s = (ssize_t)MIN;    "-9223372036854775808"
s = 'x';                         "x"      s = (size_t)MAX;     "18446744073709551615"
s = "abc";                       "abc"    s = 5.5;             "5.5"
s = 42hh;     // signed char     "42"     s = 5.5L;            "5.5"
s = 42h;      // short int       "42"     s = 5.5+3.4i;        "5.5+3.4i"
s = 0xff;                        "255"    s = 5.5L+3.4Li;      "5.5+3.4i"
```

Conversions can be explicitly specified using a compound literal.

```
s = (string){ 5 };   s = (string){ "abc" };   s = (string){ 5.5 };
```

Conversions from string to **char** * attempt to be safe. The overloaded strncpy function is safe, if the length of the C string is correct. The assignment operator and constructor both allocate storage and return its address, meaning the programmer must free it. A C string is always null terminated, implying storage is always necessary for the null.

```
string s = "abcde";
char cs[4];
strncpy( cs, s, sizeof(cs) );          "abc\0", in place
char * cp = s;       // ownership      "abcde\0", malloc
delete( cp );
cp = s + ' ' + s;    // ownership      "abcde abcde\0", malloc
delete( cp );
```

5.2.2 Length

The len operation (short for strlen) returns the length of a C (not including the terminating null) or C∀ string. For compatibility, an overloaded strlen works with C∀ strings.

```
i = len( "" );          0
i = len( "abc" );       3
i = len( cs );          3
i = strlen( cs );       3
i = len( name );        4
i = strlen( name );     4
```

5.2.3 Comparison Operators

The binary relational, <, <=, >, >=, and equality, ==, !=, operators compare C∀ strings using lexicographical ordering, where longer strings are greater than shorter strings. In C, these operators compare the C string pointer, not its value, which does not match programmer expectation. C strings use function strcmp to lexicographically compare the string value. Java has the same issue with == (reference) and .equals (value) comparison.

5.2.4 Concatenation

The binary operators + and += concatenate C **char**, **char** * and C∀ strings, creating the sum of the characters.

```
s = "";                 |  s = "";                          |  s = "";
s = 'a' + 'b';  "ab"    |  s = 'a' + 'b' + s;  "ab"         |  s = s + 'a' + 'b';  "ab"
s = 'a' + "b";  "ab"    |  s = 'a' + 'b' + s;  "abab"       |  s = s + 'a' + "b";  "abab"
s = "a" + 'b';  "ab"    |  s = 'a' + "b" + s;  "abababab"   |  s = s + "a" + 'b';  "abababab"
s = "a" + "b";  "ab"    |  s = "a" + 'b' + s;  "ababababab" |  s = s + "a" + "b";  "ababababab"
```

However, including <string.hfa> can result in ambiguous uses of the overloaded + operator.[2] For example, subtracting characters or pointers has valid use-cases:

```
ch - '0'                // find character offset
cs - cs2;               // find pointer offset
```

addition is less obvious

```
ch + 'b'                // add character values
cs + 'a';               // move pointer cs['a']
```

There are legitimate use cases for arithmetic with **signed/unsigned** characters (bytes), and these types are treated differently from **char** in C++ and C∀. However, backwards compatibility makes it impossible to restrict or remove addition on type **char**. Similarly, it is impossible to restrict or remove addition on type **char** * because (unfortunately) it is subscripting: cs + 'a' implies cs['a'] or 'a'[cs].

[2]Combining multiple packages in any programming language can result in name clashes or ambiguities.

The prior C∀ concatenation examples show complex mixed-mode interactions among **char**, **char** *, and string constants work correctly (string variables are the same). The reason is that the C∀ type-system handles this kind of overloading well using the left-hand assignment-type and complex conversion costs. Hence, the type system correctly handles all uses of addition (explicit or implicit) for **char** *.

```
printf( "%s %s %s %c %c\n", "abc", cs, cs + 3, cs['a'], 'a'[cs] );
```

Only **char** addition can result in ambiguities, and only when there is no left-hand information.

```
ch = ch + 'b';              // LHS disambiguate, add character values
s = 'a' + 'b';              // LHS disambiguate, concatenate characters
printf( "%c\n", 'a' + 'b' ); // no LHS information, ambiguous
printf( "%c\n", (return char)('a' + 'b') ); // disambiguate with ascription cast
```

The ascription cast, (**return** T), disambiguates by stating a (LHS) type to use during expression resolution (not a conversion). Fortunately, character addition without LHS information is rare in C/C∀ programs, so repurposing the operator + for string types is not a problem.

Interestingly, C++ cannot support this generality because it does not use the left-hand side of assignment in expression resolution. While it can special-case some combinations:

```
s = 'a' + s;                // compiles in C++
s = "a" + s;
```

it cannot generalize to any number of steps:

```
s = 'a' + 'b' + s;          // does not compile in C++
s = "a" + "b" + s;
```

5.2.5 Repetition

The binary operators * and *= repeat a string N times. If $N = 0$, a zero-length string results, "".

```
s = 'x' * 0;                "
s = 'x' * 3;                "xxx"
s = "abc" * 3;              "abcabcabc"
s = ("MIKE" + ' ') * 3;     "MIKE MIKE MIKE "
```

Like concatenation, there is a potential ambiguity with multiplication of characters; multiplication of pointers does not exist in C.

```
ch = ch * 3;                // LHS disambiguate, multiply character value
s = 'a' * 3;                // LHS disambiguate, concatenate characters
printf( "%c\n", 'a' * 3 );  // no LHS information, ambiguous
printf( "%c\n", (return char)('a' * 3) ); // disambiguate with ascription cast
```

Fortunately, character multiplication without LHS information is even rarer than addition, so repurposing the operator * for string types is not a problem.

5.2.6 Substring

The substring operation returns a subset of a string starting at a position in the string and traversing a length, or matching a pattern string.

length **pattern**

```
string name = "PETER"

s = name( 0, 4 );       "PETE"
s = name( 1, 4 );       "ETER"
s = name( 2, 4 );       "TER"    // clip length to 3
s = name( 4, −2 );      "ER"
s = name( 8, 2 );       ""       // clip, beyond right
s = name( 0, −2 );      ""       // clip, beyond left
s = name( −1, −2 );     "ER"
s = name( −3 );         "TER"    // to end of string
```

```
s = name( "ET" );    "ET"
s = name( "WW" );    ""  // does not occur
```

For the length form, a negative starting position is a specification from the right end of the string. A negative length means that characters are selected in the opposite (right to left) direction from the starting position. If the substring request extends beyond the beginning or end of the string, it is clipped (shortened) to the bounds of the string. If the substring request is completely outside of the original string, a null string is returned. For the pattern-form, it returns the pattern string if the pattern matches or a null string if the pattern does not match. The usefulness of this mechanism is discussed next.

The substring operation can appear on the left side of assignment, where it defines a replacement substring. The length of the right string may be shorter, the same, or longer than the length of left string. Hence, the left string may decrease, stay the same, or increase in length.

```
digit( 3, 3 ) = "";          "0126789"
digit( 4, 3 ) = "xyz";       "0126xyz"
digit( 7, 0 ) = "***";       "0126xyz"
digit(−4, 3 ) = "$$$";       "012$$$z"
digit( 5 ) = "LLL";          "012$$LLL"
```

Here, substring pattern matching is useful on the left-hand side of assignment.

```
digit( "$$" ) = "345";       "012345LLL"
digit( "LLL") = "6789";      "0123456789"
```

Supporting a regular-expression pattern is a possible extension.

The replace operation extends substring to substitute all occurrences.

```
s = replace( "PETER", "E", "XX" );       "PXXTXXR"
s = replace( "PETER", "ET", "XX" );      "PXXER"
s = replace( "PETER", "W", "XX" );       "PETER"
```

The replacement is done left-to-right and substituted text is not examined for replacement.

5.2.7 Searching

The find operation returns the position of the first occurrence of a key in a string. If the key does not appear in the string, the length of the string is returned.

```
i = find( digit, '3' );          3
i = find( digit, "45" );         4
i = find( digit, "abc" );        10  // not found, string length
```

A character-class operation indicates if a string is composed completely of a particular class of characters, *e.g.*, alphabetic, numeric, vowels, *etc*.

```
charclass vowels{ "aeiouy" };
i = include( "aabiuyoo", vowels );      2  // b non-compliant
i = include( "aaeiuyoo", vowels );      8  // compliant, string length
```

vowels defines a character class and function include checks if all characters in the string appear in the class (compliance). The position of the first non-compliant character is returned or the position of the last character if the string is compliant. There is no relationship between the order of characters in the two strings. The function exclude is the reverse of include, checking if all characters in the string are excluded from the class (compliance).

```
i = exclude( "cdbfghmk", vowels );      8  // compliant, string length
i = exclude( "cdyfghmk", vowels );      2  // y non-compliant
```

Both forms can return the longest substring of compliant characters.

```
s = include( "aabiuyoo", vowels );      "aaeiuyoo"
s = include( "aaeiuyoo", vowels );      "aa"
s = exclude( "cdbfghmk", vowels );      "cdbfghmk"
s = exclude( "cdyfghmk", vowels );      "cd"
```

There are also versions of include and exclude, returning a position or string, taking a validation function, like one of the C character-class functions.[3]

```
i = include( "1FeC34aB", isxdigit );    8  // compliant
i = include( ".,;'!\"", ispunct );      6  // compliant
i = include( "XXXx", isupper );         3  // non-compliant
```

These operations perform an *apply* of the validation function to each character, where the function returns a boolean indicating a stopping condition for the search. The position of the first non-compliant character is returned or the position of the last character if the string is compliant (and vice versa for exclude).

The translate operation returns a string with each character transformed by one of the C character transformation functions.

```
s = translate( "abc", toupper );                    "ABC"
s = translate( "ABC", tolower );                    "abc"
int tospace( int c ) { return isspace( c ) ? ' ' : c; }
s = translate( "X X\tX\nX", tospace );              "X X X X"
```

[3]It is part of the hereditary of C that these function take and return an **int** rather than a **bool**, which affects the function type.

C++

```
for ( ;; ) {
    string::size_type posn = line.find_first_of( alpha );
  if ( posn == string::npos ) break;
    line = line.substr( posn );
    posn = line.find_first_not_of( alpha );
    if ( posn != string::npos ) {
        cout << line.substr( 0, posn ) << endl;
        line = line.substr( posn );
    } else {
        cout << line << endl;
        line = " ";
    }
}
```

CFA

```
for () {
    size_t posn = exclude( line, alpha );
  if ( posn == len( line ) ) break;
    line = line( posn );
    posn = include( line, alpha );

    sout | line( 0, posn );
    line = line( posn );




}
```

Figure 5.2: Simplification by CFA API for extracting words from a line of text

5.2.8 Returning N on Search Failure

Some of the prior string operations are composite, *e.g.*, string operations returning the longest substring of compliant characters (include) are built using a search and then substring the appropriate text. However, string search can fail, which is reported as an alternate search outcome, possibly an exception. Many string libraries use a return code to indicate search failure, with a failure value of 0 or -1 (PL/I [23] returns 0). This semantics leads to an awkward pattern, which can appear many times in a string library or user code.

```
i = exclude( s, alpha );
if ( i != -1 ) return s( 0, i );
else return " ";
```

The problem is that substring does the wrong thing or fails with the failure return-code in most string libraries, so it has to be special cased.

CFA adopts a return code but the failure value is taken from the index-of function in APL [30], which returns the length of the target string N (or $N+1$ for 1 origin). This semantics allows many search and substring functions to be written without conditions, *e.g.*:

```
string include( const string & s, int (*f)( int ) ) { return s( 0, include( s, f ) ); }
string exclude( const string & s, int (*f)( int ) ) { return s( 0, exclude( s, f ) ); }
```

In string systems with an $O(1)$ length operator, checking for failure is low cost.

```
if ( include( line, alpha ) == len( line ) ) ... // not found, 0 origin
```

Figure 5.2 compares C++ and CFA string code for extracting words from a line of text, repeatedly removing non-word text and then a word until the line is empty. The CFA code is simpler solely because of the choice for indicating search failure. (A simplification of the C++ version is to concatenate a sentinel character at the end of the line so the call to find_first_not_of does not fail.)

5.2.9 C Compatibility

To ease conversion from C to C∀, C∀ provides companion C string functions. Hence, it is possible to convert a block of C string operations to C∀ strings just by changing the type **char** * to string.

```
char s[32]; // changing s's type to string works
strlen( s );                    strcpy( s, "abc" );
strnlen( s, 3 );                strncpy( s, "abcdef", 3 );
strcmp( s, "abc" );             strcat( s, "xyz" );
strncmp( s, "abc", 3 );         strncat( s, "uvwxyz", 3 );
```

However, the conversion fails with I/O, because printf cannot print a string using format code %s, because C∀ strings are not null terminated. Nevertheless, this capability does provide a useful starting point for conversion to safer C∀ strings.

The functions above provide for conversion from C **char** * into the C∀ string when s is declared to be of type string. Furthermore, the gamut of initializations and assignments already seen provides conversion from **char** * seamlessly.

```
string s1 = "abc";
s1 += "def";
```

In the other direction, i.e. C∀ into C, the analogous conversions occur, with implied allocations (for pointers), and the familiar copying vocabulary works (for buffers).

```
char * cp = s1; // allocates
strcpy( s, s1 ); // into char s[32] above, no allocation
strncpy( s, s1, 3 );
strcat( s, s1 );
strncat( s, s1, 3 );
```

The ability to initialize **char*** from string provides a compatibility bridge, albeit with some limitation. It lets a C codebase transition to C∀, even with its own functions that expect **char** *, without having to swap the whole codebase to string at once. Instead, adapting at call sites is an option.

The limitation is precisely the tradeoff that the *C∀-native* string representation is not null-terminated. So, explicit allocation management is required at these C∀-to-C call boundaries. Also, this scheme incurs runtime copying.

5.2.10 I/O Operators

The ability to input and output a string is as essential as for any other type. The goal for character I/O is to work with groups rather than individual characters. A comparison with C++ string I/O is presented as a counterpoint to C∀ string I/O.

The C++ output << and input >> operators are defined on type string. C++ output for **char**, **char** *, and string are similar. The C++ manipulators are setw, and its associated width controls left, right and setfill.

```
string s = "abc";
cout << setw(10) << left << setfill( 'x' ) << s << endl;        "abcxxxxxxx"
```

The C∀ input/output operator | is defined on type string. C∀ output for **char**, **char** *, and string are similar. The C∀ manipulators are bin, oct, hex, wd, and its associated width control and left.

```
string s = "abc";
sout | bin( s ) | nl            "0b1100001 0b1100010 0b1100011"
     | oct( s ) | nl            "0141 0142 0143"
     | hex( s ) | nl            "0x61 0x62 0x63"
     | wd( 10, s ) | nl         "       abc"
     | wd( 10, 2, s ) | nl      "        ab"
     | left( wd( 10, s ) );     "abc       "
```

C++ setfill is not considered an important string manipulator.

C++ input matching for **char**, **char** *, and string are similar, where *all* input characters are read from the current point in the input stream to the end of the type size, format width, whitespace, end of line (`'\n'`), or end of file. The C++ manipulator is setw to restrict the size. Reading into a **char** is safe as the size is 1, **char** * is unsafe without using setw to constraint the length (which includes `'\0'`), string is safe as its grows dynamically as characters are read.

```
char ch, c[10];
string s;
cin >> ch >> setw( 5 ) >> c  >> s;        'a' "bcde" "fg"
abcde   fg
```

Input text can be *gulped*, including whitespace, from the current point to an arbitrary delimiter character using getline.

C++

```
string s1, s2, s3;
getline( cin, s1, 'a' );
getline( cin, s2, 'w' );
getline( cin, s3 );
cout << s1 << ' ' << s2 << ' ' << s3 << endl;
bbbad ddwxyz
```

C∀

```
sin | getline( s1, 'a' )
    | getline( s2, 'w' )
    | getline( s3 );
sout | s1 | s2 | s3;     "bbb" "d dd" "xyz"
```

The C∀ philosophy for input is that, for every constant type in C, these constants should be usable as input. For example, the complex constant 3.5+4.1i can appear as input to a complex variable. C∀ input matching for **char**, **char** *, and string are similar. C-strings may only be read with a width field, which should match the string size. Certain input manipulators support a scanset, which is a simple regular expression from printf. The C∀ manipulators for these types are wdi,[4] and its associated width control and left, quote, incl, excl, and getline.

[4]Due to an overloading issue in the type-resolver, the input width name must be temporarily different from the output, wdi versus wd.

```
char ch, c[10];
string s;
sin | ch | wdi( 5, c ) | s;                                          'a' "bcde" "fg"
abcde fg
sin | quote( ch ) | quote( wdi( sizeof(c), c ) ) | quote( s, '[', ']' ) | nl;    'a' "bcde" "fg"
'a'  "bcde"    [fg]
sin | incl( "a-zA-Z0-9 ?!&\n", s ) | nl;                           "x?&000xyz TOM !"
x?&000xyz TOM !.
sin | excl( "a-zA-Z0-9 ?!&\n", s );                                "<>{}{}"
<>{}{}STOP
```

Note the ability to read in quoted strings with whitespace to match with program string constants. The nl at the end of an input ignores the rest of the line.

5.2.11 Assignment

While Figure 5.1, p. 106 emphasizes cross-language similarities, it elides many specific operational differences. For example, the C++ replace function selects a substring in the target and substitutes it with the source string, which can be smaller or larger than the substring. C++ modifies the mutable receiver object, replacing by position (zero origin) and length.

```
string s1 = "abcde";
s1.replace( 2, 3, "xy" );                "abxy"
```

Java cannot modify the receiver (immutable strings) so it returns a new string, replacing by text.

```
String s = "abcde";
String r = s.replace( "cde", "xy" );     "abxy"
```

Java also provides a mutable StringBuffer, replacing by position (zero origin) and length.

```
StringBuffer sb = new StringBuffer( "abcde" );
sb.replace( 2, 5, "xy" );                 "abxy"
```

However, there are anomalies. StringBuffer's substring returns a String copy that is immutable rather than modifying the receiver. As well, the operations are asymmetric, *e.g.*, String has replace by text but not replace by position and vice versa for StringBuffer.

More significant operational differences relate to storage management, often appearing through assignment (target = source), and are summarized in Table 5.1, which defines properties type abstraction, state, symmetry, and referent. The following discussion justifies the table's yes/no entries per language.

In C, these declarations are very different.

```
char x[] = "abcde";
char * y = "abcde";
```

Both associate the declared name with the fixed, six contiguous bytes: {'a', 'b', 'c', 'd', 'e', 0}. But x is allocated on the stack (with values filled at the declaration), while y refers to the executable's read-only data-section. With x representing an allocation, it offers information in **sizeof**(x) that y does not. But this extra information is second-class, as it can only be used in the immediate lexical context, *i.e.*, it cannot be passed to string operations or user functions. Only pointers to text buffers are first-class, and discussed further.

Table 5.1: Comparison of languages' strings, storage management perspective

			Supports Helpful?			
	Required	Helpful	C	C++	Java	C∀
Type Abstraction	Low-level: The string type is a varying amount of text communicated via a parameter or return.	High-level: The string type relieves the user of managing memory for the text.	no	yes	yes	yes
State	Fast Initialize: The target receives the characters of the source without copying the characters, resulting in an Alias or Snapshot.	Alias: The target variable names the source text; changes made in either variable are visible in both.	yes	yes	no	yes
		Snapshot: Subsequent operations on either the source or target variable do not affect the other.	no	no	yes	yes
Symmetry	Laxed: The target's type is anything string-like; it may have a different status concerning ownership.	Strict: The target's type is the same as the source; both strings are equivalent peers concerning ownership.	N/A	no	yes	yes
Referent	Variable-Constrained: The target can accept the entire text of the source.	Fragment: The target can accept an arbitrary substring of the source.	no	no	yes	yes

Notes

- All languages support Required in all criteria.
- A language gets "Supports Helpful" in one criterion if it can do so without sacrificing the Required achievement on all other criteria.
- The C "string" is **char** *, under the conventions of <string.h>. Because this type does not manage a text allocation, symmetry does not apply.
- The Java String class is analyzed; its StringBuffer class behaves similarly to C++.

```
char * s = "abcde";
char * s1 = s;                  // alias state, N/A symmetry, variable-constrained referent
char * s2 = &s[1];              // alias state, N/A symmetry, variable-constrained referent
char * s3 = &s2[1];             // alias state, N/A symmetry, variable-constrained referent
printf( "%s %s %s %s\n", s, s1, s2, s3 );
abcde abcde bcde cde
```

All of these aliased strings rely on the single null termination character at the end of s. The issue of symmetry does not apply to a low-level API because no management of a text buffer is provided. With the **char** * type not managing the text storage, there is no ownership question, *i.e.*, operations on s1 or s2 never lead to their memory becoming reusable. While s3 is a valid C-string that contains a proper substring of s1, the s3 technique does not constitute having a fragment referent because null termination implies the substring cannot be chosen arbitrarily; the technique works only for suffixes.

In C++, string offers a high-level abstraction.

```
string s = "abcde";
string & s1 = s;                    // alias state, lax symmetry, variable-constrained referent
string s2 = s;                      // no fast-initialize (strict symmetry, variable-constrained referent)
string s3 = s.substr( 1, 2 );       // no fast-initialize (strict symmetry, fragment referent)
string s4 = s3.substr( 1, 1 );      // no fast-initialize (strict symmetry, fragment referent)
cout << s << ' ' << s1 << ' ' << s2 << ' ' << s3 << ' ' << s4 << endl;
abcde abcde abcde bc c
string & s5 = s.substr(2,4);        // error: cannot point to temporary
```

The s1 lax symmetry reflects how its validity depends on the lifetime of s. It is common practice in C++ to use the s1-style for a by-reference function parameter. Doing so assumes that the callee only uses the referenced string for the duration of the call, *i.e.*, no storing the parameter (as a reference) for later. So, when the called function is a constructor, its definition typically uses an s2-style copy-initialization. Exceptions to this pattern are possible, but require the programmer to assure safety where the type system does not. The s3 initialization must copy the substring to support a subsequent c_str call, which provides null-termination, generally at a different position than the source string's. s2 assignment could be made fast, by reference-counting the text area and using copy-on-write, but would require an implementation upgrade.

In Java, String also offers a high-level abstraction:

```
String s = "abcde";
String s1 = s;                      // snapshot state, strict symmetry, variable-constrained referent
String s2 = s.substring( 1, 3 );    // snapshot state (possible), strict symmetry, fragment referent
String s3 = s2.substring( 1, 2 );   // snapshot state (possible), strict symmetry, fragment referent
System.out.println( s + ' ' + s1 + ' ' + s2 + ' ' + s3 );
abcde abcde bc c
System.out.println( (s == s1) + "  " + (s == s2) + "  " + (s2 == s3) );
true false false
```

Note that Java's substring takes a start and end position, rather than a start position and length. Here, facts about Java's implicit pointers and pointer equality can over complicate the picture, and so are ignored. Furthermore, Java's string immutability means string variables behave as simple values. The result in s1 is the pointer in s, and their pointer equality confirm no time is spent copying characters. With s2, the case for fast-copy is more subtle. Certainly, its value is not pointer-equal to s, implying at least a further allocation. But because Java is *not* constrained to use a null-terminated representation, a standard-library implementation is free to refer to the source characters in-place. Java does not meet the aliasing requirement because immutability makes it impossible to modify. Java's StringBuffer provides aliasing (see replace example on page 115), though without supporting symmetric treatment of a fragment referent, *e.g.*, substring of a StringBuffer is a String; as a result, StringBuffer scores as C++. The easy symmetry that the Java string enjoys is aided by Java's garbage collection; Java's s1 is doing effectively the operation of C++'s s1, though without the consequence of complicating memory management.

Finally, in C∀, string also offers a high-level abstraction:

```
string s = "abcde";
string & s1 = s;                    // alias state, lax symmetry, variable-constrained referent
string s2 = s;                      // snapshot state, strict symmetry, variable-constrained referent
string s3 = s`share;                // alias state, strict symmetry, variable-constrained referent
```

```
string s4 = s( 1, 2 );              // snapshot state, strict symmetry, fragment referent
string s5 = s4( 1, 1 )`share;       // alias state, strict symmetry, fragment referent
sout | s | s1 | s2 | s3 | s4 | s5;
abcde abcde abcde abcde bc c
```

The C∀ string manages storage: it handles all assignments, including those of fragment referents with fast initialization; it provides the choice between snapshot and alias semantics; it does so symmetrically with one type (which assures text validity according to the lifecycles of the string variables). The s1 case is the same in C∀ as in C++. But the C∀ s2 delivers a fast snapshot, while the C++ s2 does not. C∀ offers the snapshot-vs-alias choice in s2-vs-s3 and s4-vs-s5. Regardless of snapshot-vs-alias and fragment-vs-whole, the target is always of the same string type.

5.2.12 Logical Overlap

It may be unfamiliar to combine Table 5.1, p. 116's alias state and fragment referent in one API, or at the same time. This section shows the capability in action.

A GUI text-editor provides a metaphor. Selecting a block of text using the mouse defines an aliased substring within a document. Typing in this area overwrites what was there, replacing the originally selected text with more or less text. But the *containing document* also grows or shrinks, not just the selection. This action models assigning to an aliased substring when one string is completely contained in the other.

Extend the metaphor to a multi-user editor. If Alice selects a range of text at the bottom, while Bob is rewriting a paragraph at the top, Alice's selection holds onto the characters initially selected, unaffected by Bob making the document grow/shrink even though Alice's start index in the document is changing. This action models assigning to an aliased substring when the two strings do not overlap.

Logically, Alice's and Bob's actions on the whole document are like two single-user-edit cases, giving the semantics of the individual edits flowing into a whole. But, there is no need to have two separate document strings. Even if a third selection removes all the text, both Alice's and Bob's strings remain. The independence of their selections assumes that the editor API does not allow the selection to be enlarged, *i.e.*, adding text from the containing environment, which may have disappeared.

This leaves the "Venn-diagram overlap" cases, where Alice's and Bob's selections overlap at the top, bottom, or corner. In this case, the selection areas are dependent, and so, changes in content and size in one may have an effect in the other. There are multiple possible semantics for this case. The remainder of this section shows the chosen semantics for all of the cases.

String sharing is expressed using the `share marker to indicate aliasing (mutations shared) *vs.* snapshot (not quite an immutable result, but one with subsequent mutations isolated). This aliasing relationship is a *sticky property* established at initialization. For example, here strings s1 and s1a are in an aliasing relationship, while s2 is in a copy relationship.

```
                          s1          s1a         s2
    string s1 = "abc";    "abc"
string s1a = s1`share;                "abc"
      string s2 = s1a;                            "abc"
```

Here, the aliasing (`share) causes partial changes (subscripting) to flow in both directions. (In the following examples, note how s1 and s1a change together, and s2 is independent.)

	s1	s1a	s2
initial	"abc"	"abc"	"abc"
s1[1] = '+';	"a+c"	"a+c"	"abc"
s1a[1] = '-';	"a-c"	"a-c"	"abc"
s2[1] = '|';	"a-c"	"a-c"	"a|c"

Similarly for complete changes.

	s1	s1a	s2
initial	"a-c"	"a-c"	"a|c"
s1 = "qrs";	"qrs"	"qrs"	"a|c"
s1a = "tuv";	"tuv"	"tuv"	"a|c"
s2 = "wxy";	"tuv"	"tuv"	"wxy"

Because string assignment copies the value, RHS aliasing is irrelevant. Hence, aliasing of the LHS is unaffected.

	s1	s1a	s2
initial	"tuv"	"tuv"	"wxy"
s1 = s2;	"wxy"	"wxy"	"wxy"
s1 = "aaa";	"aaa"	"aaa"	"wxy"
s2 = s1;	"aaa"	"aaa"	"aaa"
s2 = "bbb";	"aaa"	"aaa"	"bbb"
s2 = s1a;	"aaa"	"aaa"	"aaa"
s2 = "ccc";	"aaa"	"aaa"	"ccc"
s1 = "xxx";	"xxx"	"xxx"	"ccc"

Now, consider string s1_mid being an alias in the middle of s1, along with s2, made by a copy from the middle of s1.

	s1	s1_mid	s2
string s1 = "abcd";	"abcd"		
string s1_mid = s1(1,2)`share;		"bc"	
string s2 = s1_mid(0,2);			"bc"

Again, `share passes changes in both directions; copy does not. As a result, the index values for the position of b are 1 in the longer string "abcd" and 0 in the shorter aliased string "bc". This alternate positioning also applies to subscripting.

	s1	s1_mid	s2
initial	"abcd"	"bc"	"bc"
s1 [1] = '+';	"a+cd"	"+c"	"bc"
s1_mid[0] = '-';	"a-cd"	"-c"	"bc"
s2 [0] = '|';	"a-cd"	"-c"	"|c"

Finally, assignment flows through the aliasing relationship without affecting its structure.

	s1	s1_mid	s2
initial	"a-cd"	"-c"	"|c"
s1_mid = "ff";	"affd"	"ff"	"|c"
s2 = "gg";	"affd"	"ff"	"gg"

In the "ff" assignment, the result is straightforward to accept because the flow direction is from contained (small) to containing (large). The following rules explain aliasing substrings that flow in the opposite direction, large to small.

Growth and shrinkage are natural extensions, as for the text-editor analogy, where an empty substring is as real as an empty string.

	s1	s1_mid
initial	"affd"	"ff"
s1_mid = "hhhh";	"ahhhhd"	"hhhh"
s1_mid = "i";	"aid"	"i"
s1_mid = "";	"ad"	""
s1_mid = "jj";	"ajjd"	"jj"

Multiple portions of a string can be aliased.

	s1	s1_bgn	s1_mid	s1_end
initial	"ajjd"		"jj"	
string s1_bgn = s1(0, 1)`share;		"a"		
string s1_end = s1(3, 1)`share;				"d"
s1_bgn = "zzzz";	"zzzzjjd"	"zzzz"	"jj"	"d"

When s1_bgn's size increases by 3, s1_mid's starting location moves from 1 to 4 and s1_end's from 3 to 6,

Changes can happen on an aliasing substring that overlaps.

	s1	s1_bgn	s1_crs	s1_mid	s1_end
initial	"zzzzjjd"	"zzzz"		"jj"	"d"
string s1_crs = s1(3, 2)`share;			"zj"		
s1_crs = "+++";	"zzz+++jd"	"zzz"	"+++"	"j"	"d"

Strings s1_crs and s1_mid overlap at character 4, 'j', because the substrings are 3,2 and 4,2. When s1_crs's size increases by 1, s1_mid's starting location moves from 4 to 5, but the overlapping character remains, changing to '+'.

Figure 5.3 shows similar relationships when passing the results of substring operations by reference and by value to a subprogram. Because this example is complex, Figure 5.4, p. 122 shows line-by-line changes to the reference parameters x, a, b, c, d, and value parameter e as function test executes (read top to bottom, left to right). The assignment to variable a changes its substring "ss" to "aaa" increasing a to "aaat" (3 to 4 characters), increasing x from 11 to 12 characters, and pushing b, c and d right one character in x because there is no overlap with the changed characters. Similarly, the assignment to variable b changes its substring "tt" to "bbb" increasing b to "tbbb" (3 to 4 characters), increasing x from 12 to 13 characters, and pushing c and d right one character in x because there is no overlap with the changed characters. The assignment to variable c changes its substring "u" (at the end) to "ccc" increasing c to "tuuccc" (4 to 6

```
// x, a, b, c, & d are substring results passed by reference
// e is a substring result passed by value
void test( string & x, string & a, string & b, string & c, string & d, string e ) {
    a( 0, 2 ) = "aaa";
    b( 1, 12 ) = "bbb";
    c( 3, 5 ) = "ccc";
    c = "yyy";
    d( 0, 3 ) = "ddd";
    e( 0, 3 ) = "eee";
    x = e;
}

int main() {
    string x = "xxxxxxxxxxx";
    test( x, x(0, 3), x(2, 3), x(4, 5), x(8, 5), x(8, 5) );
}
```

x	a	b	c	d	e
"ssttttuuuww"	"sst"	"ttt"	"ttuuu"	"uww"	"uww"
"aaattttuuuww"	"aaat"	"ttt"	"ttuuu"	"uww"	"uww"
"aaatbbbtuuuww"	"aaat"	"tbbb"	"tuuu"	"uww"	"uww"
"aaatbbbtuucccww"	"aaat"	"tbbb"	"tuuccc"	"cccww"	"uww"
"aaatbbbyyyyww"	"aaat"	"tbbb"	"yyy"	"ww"	"uww"
"aaatbbbyyyyddd"	"aaat"	"tbbb"	"yyy"	"ddd"	"uww"
"aaatbbbyyyyddd"	"aaat"	"tbbb"	"yyy"	"ddd"	"uww"
"eee"	""	""	""	""	"eee"

Figure 5.3: Passing substrings as parameters, program

characters), increasing x from 13 to 15 characters, and d remains in the same position because it overlaps with the changed characters and its "u" grows to "ccc" giving "cccww". The second assignment to variable c changes its substring "tuuccc" to "yyy" decreasing c to "yyy" (6 to 3 characters), decreasing x from 15 to 12 characters, and the 3 character overlapping with d are removed leaving "ww". The assignment to variable d changes its substring "ww" to "ddd" increasing d from 2 to 3 characters, increasing x from 11 to 12 characters. The assignment to variable e changes its substring "uww" to "eee" with no change in length. Finally, the assignment to x sets it pointing at e, and all prior overlapping strings with x become empty (null) strings.

5.3 Storage Management

This section discusses the implementation of the prior string semantics, specifically, when strings overlap partially or completely.

The centrepiece of the string module is its memory manager. A string is a smart pointer into this buffer.

5.3.1 Individual String

Each string is a C structure. Its contents are pointers to other strings, pointers to global structures and count/flag data. Each field's purpose is to access or interpret data outside of the individual string structure, often shared data. So, a full understanding relies on the implementation descriptions in the subsequent sections.

```
function entry
x 0 1 2 3 4 5 6 7 8 9 10  e 0 1 2
  s s t t t t u u u w w    u w w
a 0 1 2
  s s t
b    0 1 2
     t t t
c       0 1 2 3 4
        t t u u u
d             0 1 2
              u w w
```

```
          a( 0, 2 ) = "aaa";
x 0 1 2 3 4 5 6 7 8 9 10 11  e 0 1 2
  a a a t t t t u u u w w     u w w
a 0 1 2 3
  a a a t
b      0 1 2
       t t t
c         0 1 2 3 4
          t t u u u
d               0 1 2
                u w w
```

```
          b( 1, 12 ) = "bbb";
x 0 1 2 3 4 5 6 7 8 9 10 11 12  e 0 1 2
  a a a t b b b t u u u w w     u w w
a 0 1 2 3
  a a a t
b      0 1 2 3
       t b b b
c            0 1 2 3
             t u u u
d                 0 1 2
                  u w w
```

```
          c( 3, 5 ) = "ccc";
x 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14  e 0 1 2
  a a a t b b b t u u c c c w w     u w w
a 0 1 2 3
  a a a t
b      0 1 2 3
       t b b b
c            0 1 2 3 4 5
             t u u c c c
d                 0 1 2 3 4
                  c c c w w
```

```
          c = "yyy";
x 0 1 2 3 4 5 6 7 8 9 10 11  e 0 1 2
  a a a t b b b y y y w w     u w w
a 0 1 2 3
  a a a t
b      0 1 2 3
       t b b b
c            0 1 2
             y y y
d                 0 1
                  w w
```

```
          d( 0, 3 ) = "ddd";
x 0 1 2 3 4 5 6 7 8 9 10 11 12  e 0 1 2
  a a a t b b b y y y d d d     u w w
a 0 1 2 3
  a a a t
b      0 1 2 3
       t b b b
c            0 1 2
             y y y
d                 0 1 2
                  d d d
```

```
          e( 0, 3 ) = "eee";
x 0 1 2 3 4 5 6 7 8 9 10 11 12  e 0 1 2
  a a a t b b b y y y d d d     e e e
a 0 1 2 3
  a a a t
b      0 1 2 3
       t b b b
c            0 1 2
             y y y
d                 0 1 2
                  d d d
```

```
          x = e;
        x 0 1 2   e 0 1 2
          e e e     e e e
        a
          ""
        b
          ""
        c
          ""
        d
          ""
```

Figure 5.4: Passing substrings as parameters, execution

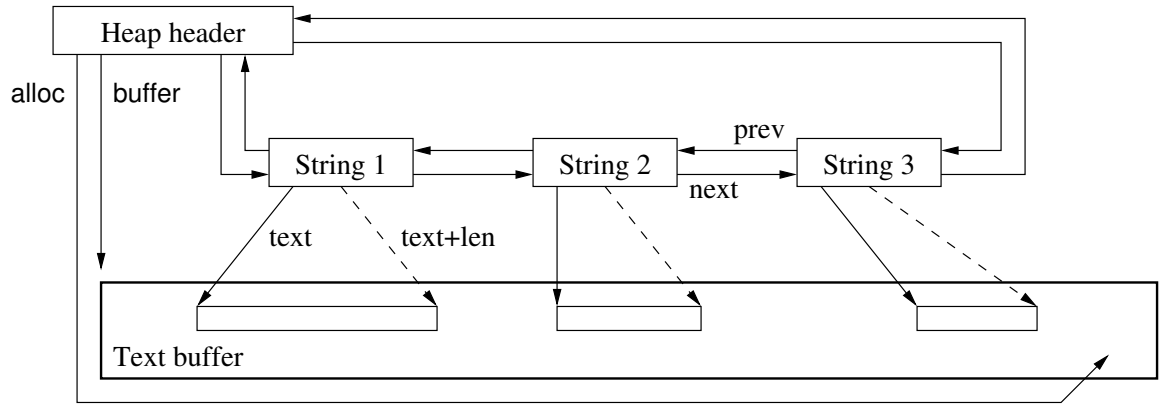


Figure 5.5: String memory-management data structures

As a summary, however, the field list is:

- Intrusive links for the heap-inventory list. Two pointers. The heap inventory shadows the live allocations of fragments of a shared text buffer, discussed in Section 5.3.2, diagrammed in Figure 5.5. Each distinct heap, per Section 5.3.5, p. 129, defines a distinct inventory list; a string with sharing disabled is in a heap by itself.
- Heap header of Figure 5.5. One pointer. All strings in the heap inventory refer to the same header.
- Text-buffer allocation. One pointer. Refers to first byte of this string's text.
- Number of bytes in this string. size_t. Length of the shared text buffer that is in use.
- Intrusive links for neighbouring strings in a Share-Edit Set (SES). Two pointers. The SES is explained in Section 5.3.4, p. 128. It is a subset of the heap inventory.
- Flag tracking whether the heap (of inventory neighbours and header) is allocated exclusively for the current SES. Boolean. Set for strings with sharing disabled (per Section 5.3.5, p. 129), in which case, deleting the last SES member deletes the heap. Cleared for strings that share, *i.e.*, where the heap should outlive the SES.

5.3.2 String Heap

The management scheme defines a shared buffer for string text. Allocation in this buffer is always via a bump-pointer; the buffer is compacted and/or relocated (to grow) when it fills.

This cycle of frequent cheap allocations, interspersed with infrequent expensive compactions, has obvious similarities to a general-purpose memory-manager based on garbage collection (GC). A few differences are noteworthy. First, in a general purpose manager, the allocated objects may contain pointers to other objects, making the transitive reachability of these objects a crucial property. Here, the allocations are text, so one allocation never keeps another alive. Second, in a general purpose manager, the handle that keeps an allocation alive is a bare pointer. For strings, a fatter representation is acceptable because this pseudo-pointer is only used for entry into the string-heap, not for general data-substructure linking around the general heap.

Figure 5.5 shows the representation. The heap header and text buffer define a sharing context. Normally, one global context is appropriate for an entire program; concurrency is discussed in Section 5.3.5, p. 129. A string is a handle to a node in a linked list containing information about string text in the buffer. The list is doubly linked for $O(1)$ insertion and removal at any location. Strings are ordered in the list by text start address. The heap header maintains a next-allocation pointer, alloc, pointing to the last live allocation in the buffer. No external references point into the buffer and the management procedure relocates the text allocations as needed. A string handle references a containing string, while its string is contiguous and not null terminated. The length sets an upper limit on the string size, but is large (4 or 8 bytes). String handles can be allocated in the stack or heap, and represent the string variables in a program. Normal C life-time rules apply to guarantee correctness of the string linked-list. The text buffer is large enough with good management so that often only one dynamic allocation is necessary during program execution, but not so large as to cause program bloat.

When the text buffer fills, *i.e.*, the next new string allocation causes alloc to point beyond the end of the buffer, the strings are compacted. The linked handles define all live strings in the buffer, which indirectly defines the allocated and free space in the buffer. The string handles are maintained in sorted order, so the handle list can be traversed, copying the first live text to the start of the buffer, and subsequent strings after each other. After compaction, if free storage is still less than the new string allocation, a larger text buffer is heap-allocated, the current buffer is copied into the new buffer, and the original buffer is freed. Note that the list of string handles is structurally unaffected during a compaction; only the text pointers in the handles are modified to new buffer locations.

Object lifecycle events are the *subscription-management* triggers in such a service. There are two fundamental string-creation functions: importing external text like a C-string or reading a string, and initialization from an existing C∀ string. When importing, storage comes from the end of the buffer, into which the text is copied. To maintain sorted order, the new string handle is inserted at the end of the handle list because the new text is at the end of the buffer. When initializing from text already in the buffer, the new handle is a second reference into the original run of characters. To maintain sorted order, the new handle's linked-list position is after the original handle. Both string initialization styles preserve the string module's internal invariant that the linked-list order matches the buffer order. For string destruction, handles are removed from the list. Once the last handle using a run of buffer characters is destroyed, that buffer space is excluded from use until the next compaction.

Certain string operations result in a substring of another string. The resulting handle is then placed in the correct sorted position in the list, possibly requiring a short linear search to locate the position. For string operations resulting in a new string, that string is allocated at the end of the buffer. For shared-edit strings, handles that originally reference containing locations need to see the new value at the new buffer location. These strings are moved to appropriate locations at the end of the list (see details in Section 5.3.4, p. 128). For nonshared-edit strings, a containing string can be moved and the other strings can remain in the same position. String assignment works similarly to string initialization, maintaining the invariant of linked-list order matching buffer order.

At the level of the memory manager, these modifications can always be explained as assignment and appendment. For example, an append is an assignment into the empty substring at the

end of the buffer. Favourable conditions allow for in-place editing: where there is room for the resulting value in the original buffer location, and where all handles referring to the original buffer location see the new value. One notable example of favourable conditions occurs because the most recently written string is often the last in the buffer, after which a large amount of free space occurs. Now, repeated appends (reading) can occur without copying previously accumulated characters. However, the general case requires a new buffer allocation: where the new value does not fit in the old place, or if other handles are still using the old value.

5.3.3 RAII Limitations

C∀ offers a system of constructors and destructors that is quite comparable to C++'s, outside of a particular corner case, where there is a problem. The problem is fixable, but a fix is beyond the C∀ team's current capacity. Unfortunately for strings, the simultaneous perspectives of, "A string is a handle into a complex allocation system," and, "A string is just a value," combine to exercise the problematic corner case. So, the C∀ string library works around the problem, through a temporary mechanism that incurs a performance penalty. Where this penalty is unacceptable, the workaround mechanism can be disabled, for a sacrifice in user-code simplicity. This section develops a precise accounting of the general problem, its eventual general solution, why strings specifically encounter the problem, the string workaround, how to avoid the workaround and the roadmap implications for string-using code.

Earlier work on C∀ [47, ch. 2] implemented object constructors and destructors for all types (basic and user defined). A constructor is a user-defined function run implicitly *after* an object's storage is allocated, and a destructor is a user-defined function run *before* an object's storage is deallocated. This feature, called Resource Acquisition Is Initialization (RAII) [51, p. 389], helps guarantee invariants for users before accessing an object and for the programming environment after an object terminates.

The purposes of these invariants goes beyond ensuring authentic values inside an object. Invariants can also track data about managed objects in other data structures. Reference counting is an example application of an invariant outside of the immediate data value. With a reference-counting smart-pointer, the constructor and destructor *of a pointer type* track the lifecycle of the object it points to. Both C++ and C∀ RAII systems are powerful enough to achieve reference counting.

In general, a lifecycle function has access to an object by location, *i.e.*, constructors and destructors receive a parameter providing an object's memory address.[5]

```
struct S { int * ip; };
void ?{}( S & this ) { this.ip = new(); }          // default constructor
void ?{}( S & this, int i ) { this{}; *this.ip = i; } // initializing constructor
void ?{}( S & this, S s ) { this{ *s.ip }; }       // copy constructor
void ^?{}( S & this ) { delete( this.ip ); }       // destructor
```

Such basic examples use the this address only to gain access to the values being managed. But lifecycle logic can use the *address*, too, *e.g.*, add the this object to a collection at creation and remove it at destruction.

[5]Historically in C++, the type of **this** is a pointer versus a reference; in C∀, the first parameter of a constructor or destructor must be a reference.

```
struct Thread {                                 // list node
    // private
    inline dlink( Thread );
};
static dlist( Thread ) ready_queue;
void ?{}( Thread & this ) { /* initialize thread */ insert_last( ready_queue, this ); }
void ^?{}( Thread & this ) { remove( this );  /* de-initialize thread */ } // implicitly use ready_queue
```

A module providing the Thread (node) type can traverse ready_queue to manipulate the objects. Hence, declaring a Thread not only ensures that it begins with an initially "good" value, but it also provides an implicit subscription to a service (scheduler) that keeps the value "good" during its lifetime. Again, both C∀ and C++ support this usage style.

A third capability concerns *implicit* copying. For passing and returning by value in function call, the parameters and return results do not require construction because after allocation the argument or return value is copied into the corresponding object. In the parameter direction, the language's function-call arranges copy-constructor calls at the control transfer into the callee. In the return direction, the roles are reversed and the copy-constructor calls happen at the return to the caller. C++ supports this capability. C∀ offers limited support; simple examples work, but implicit copying does not combine successfully with other RAII capabilities.

C++ also offers move constructors and return-value optimization [41]. These features help reduce unhelpful copy-constructor calls, which, for types like the initial S example, lead to extra memory allocations. C∀ does not currently have these features; adding similarly-intended features to C∀ is desirable. However, this section is about a problem in the realization of features that C∀ already supports. Hence, the comparison continues with the classic version of C++ that treats such copy-constructor calls as necessary.

To summarize the unsupported combinations, the relevant features are:

1. Object provider implements lifecycle functions to manage a resource outside of the object.
2. Object provider implements lifecycle functions to store references to the object, often originating from outside of it.
3. Object user expects to pass (in either direction) an object by value for function calls.

C++ supports all three simultaneously. C∀ does not currently support 2 and 3 on the same object, though 1 works along with either one of 2 or 3. C∀ needs to be fixed to support all three simultaneously.

The reason C∀ does not support 2 with 3 is a holdover from how C∀ lowered function calls to C, before C∀ got references and RAII. At that time, adhering to a principle of minimal intervention, this code was treated as a passthrough:

```
struct U { ... };
// add RAII ctor/dtor function
void f( U u ) { ... /* access fields of u */ ... }
U x;
f( x );
```

However, adding RAII constructors/destructor changes things.

Figure 5.6 shows the common C++ lowering [29, Sec. 3.1.2.3] (right) proceeds differently than the present C∀ lowering (left). The current C∀ scheme is still using a by-value C call. C does a

C∀ today

```
struct U {...};
// RAII elided
void f( U u ) {

    ... /* access fields of u */ ...

}
U x; // call default ctor
{
    U __u_for_f = x;  // call copy ctor
    f( __u_for_f );
    // call dtor, __u_for_f
}
// call dtor, x
```

C++, C∀ future

```
struct U {...};
// RAII elided
void f( U * __u_orig ) {
    U u = * __u_orig;  // call copy ctor
    ... /* access fields of u */ ...
    // call dtor, u
}
U x; // call default ctor


f( &x ) ;


// call dtor, x
```

Figure 5.6: Code lowering by RAII implementation

memcpy on structures passed by value. And so, the body of f sees the bits of __u_for_f occurring at an address that has never been presented to the U lifecycle functions. If U is trying to have a style-2 invariant, it shows up broken in f: references supposedly to u are actually to __u_for_f. The C++ scheme does not have this problem because it constructs the u copy in the correct location within f. Yet, the current C∀ scheme is sufficient to deliver style-1 invariants (in this style-3 use case) because this scheme still does the correct number of lifecycle calls, using correct values, at correct times. So, reference-counting or simple ownership applications get their invariants respected under call/return-by-value.

The string API offers style 3's pass-by-value, *e.g.*, in the return of `"a" + "b"`. Its implementation uses the style-2 invariant of the string handles being linked to each other, helping to achieve high performance. Since these two RAII styles cannot coexist, a workaround splits the API into two layers: one that provides pass-by-value, built upon the other with inter-linked handles. The layer with pass-by-value incurs a performance penalty, while the layer without delivers the desired runtime performance. The slower, friendlier High Level API (HL, type string) wraps the faster, more primitive Low Level API (LL, type string_res, abbreviating "resource"). Both APIs present the same features, up to return-by-value operations being unavailable in LL and implemented via the workaround in HL. The intention is for most future code to target HL. When the RAII issue is fixed, the full HL feature set is achievable using the LL-style lifetime management. Then, HL can be removed, LL's type renamed to string, and programs generated with the current HL will run faster. In the meantime, performance-critical sections of applications must use LL. Subsequent performance experiments (see Section 5.4, p. 131) use the LL API when comparing C∀ to other languages. This measurement is intended to estimate of the goal state for C∀. A separate measure of the HL overhead is also included. Hence, Section 5.3.2, p. 123 is describing the goal state for C∀. In the present state, the internal type string_res replaces string for an inter-linked handle.

To use LL, a programmer rewrites HL invocations that used pass-by-value APIs into LL invocations where resourcing is more explicit. Many invocations are unaffected, notably assignment and

Table 5.2: HL to LL C∀ string-API lowering

HL	LL
string s = "a" + "b";	string_res sr = "a"; sr += b;

string s = "abcde";
string s2 = s(2, 3); // *s2 == "cde"*
s(2,3) = "x"; // *s == "abx" && s2 == "cde"*
	string_res sr = "abcde";
	string_res sr2 = {sr, 2, 3}; // *sr2 == "cde"*
	string_res sr_mid = { sr, 2, 3, SHARE };
	sr_mid = "x"; // *sr == "abx" && sr2 == "cde"*

string s = "abcde"; s[2] = "xxx"; // *s == "abxxxde"*	string_res sr = "abcde"; string_res sr_mid = { sr, 2, 1, SHARE }; mid = "xxx"; // *sr == "abxxxde"*

comparison. Table 5.2 shows, of the capabilities listed in Figure 5.1, p. 106, only three cases need revisions. The actual HL workaround wraps string as a pointer to a uniquely owned, heap-allocated string_res. This arrangement has string using style-1 RAII, which is compatible with pass-by-value.

5.3.4 Sharing Implementation

The C∀ string module has two mechanisms to deal with string handles sharing text. In the first type of sharing, the user requests that both string handles be views of the same logical, modifiable string. This state is typically produced by the substring operation.

```
string s = "abcde";
string s1 = s( 1, 2 )`share;        // explicit sharing
s[1] = 'x';                          // change s and s1
sout | s | s1;
axcde xc
```

Here, the source string-handle is referencing an entire string, and the resulting, newly made, string handle is referencing a contained portion of the original. In this state, a modification made in the overlapping area is visible in both strings.

The second type of sharing happens when the system implicitly delays the physical execution of a logical *copy* operation, as part of its copy-on-write optimization. This state is typically produced by constructing a new string, using an original string as its initialization source.

```
string s = "abcde";
string s1 = s( 1, 2 );               // no sharing
s[1] = 'x';                          // copy-on-write s1
sout | s | s1;
axcde bc
```

In this state, a subsequent modification done on one handle triggers the deferred copy action, leaving the handles referencing different text within the buffer, holding distinct values.

A further abstraction helps distinguish the two senses of sharing. A share-edit set (SES) is an equivalence class over string handles, being the reflexive, symmetric and transitive closure of the relationship of one string being constructed from another, when the “share” option is given.

The SES is represented by a second linked list among the handles. A string with no sharing is in a SES by itself. Inside a SES, a modification of one substring portion may change the value in another substring portion, depending on whether the two overlap. Conversely, no value change can flow outside of a SES. Even if a modification on one string handle does not reveal itself *logically* to another handle in the same SES (because they do not overlap), if the modification is length-changing, completing the modification requires visiting the second handle to adjust its location in the sliding text.

5.3.5 Controlling Implicit Sharing

There are tradeoffs associated with sharing and its implicit copy-on-write mechanism. Several qualitative matters are detailed in Section 5.4, p. 131. In detail, string sharing has inter-linked string handles, so managing one string is also managing the neighbouring strings, and from there, a data structure of the "set of all strings." Therefore, it is useful to toggle this capability on or off when it is not providing any application benefit.

The C∀ string library provides the type string_sharectx to control an ambient sharing context. It allows two adjustments: to opt out of sharing entirely or to begin sharing within a private context. Running with sharing disabled can be thought of as a C++ STL-emulation mode, where each string is dynamically allocated. The chosen mode applies for the duration of the lifetime of the created string_sharectx object, up to being suspended by child lifetimes of different contexts. Figure 5.7 illustrates this behaviour by showing the stack frames of a program in execution. In this example, the single-letter functions are called in alphabetic order. The functions a, b and g share string character ranges with each other, because they occupy a common sharing-enabled context. The function e shares within itself (because its is in a sharing-enabled context), but not with the rest of the program (because its context is not occupied by any of the rest of the program). The functions c, d and f never share anything, because they are in a sharing-disabled context. Executing the example does not produce an interesting outcome, but the comments in the picture indicate when the logical copy operation runs with

share: the copy being deferred, as described through the rest of this section (fast), or
copy: the copy performed eagerly (slow).

Only eager copies can cross string_sharectx boundaries. The intended use is with stack-managed lifetimes, in which the established context lasts until the current function returns, and affects all functions called that do not create their own contexts.

5.3.6 Sharing and Threading

The C∀ string library provides no thread safety, the same as C++ string, providing similar performance goals. Threads can create their own string buffers and avoid passing these strings to other threads, or require that shared strings be immutable, as concurrent reading is safe. A positive consequence of this approach is that independent threads can use the sharing buffer without locking overhead. When string sharing amongst threads is required, program-wide string-management can toggled to non-sharing using malloc/free, where the storage allocator is assumed to be thread-safe. Finally, concurrent users of string objects can provide their own mutual exclusion.

```
void a( string & sm ) {
    string sm_a = sm; // share
}
void b( string & sm ) {
    string sm_b = sm; // share
}
void c( string & sm, string & sh1 ) {
    string sm_c = sm; // copy
    string sh1_c = sh1; // copy
}
void d( string & sm, string & sh1 ) {
    string sm_d = sm; // copy
    string sh1_d = sh1; // copy
}
void e( string & sm, string & sh1, string & sh2 ) {
    string sm_e = sm; // copy
    string sh1_e = sh1; // copy
    string sh2_e = sh2; // share
}
void f( string & sm, string & sh1 ) {
    string sm_f = sm; // copy
    string sh1_f = sh1; // copy
}
void g( string & sm ) {
    string sm_g = sm; // share
}
void helper2( string & sm, string & sh1 ) {
    d( sm, sh1 );
    string_sharectx ctx_on = {NEW_SHARING};
    string sh2 = "sh2";
    e( sm, sh1, sh2 );
}
void helper1( string & sm ) {
    b( sm );
    string_sharectx ctx_off = {NO_SHARING};
    string sh1 = "sh1";
    c( sm, sh1 );
    helper2( sm, sh1 );
    f( sm, sh1 );
}
int main() {
    string sm = "sm";
    a( sm );
    helper1( sm );
    g( sm );
}
```

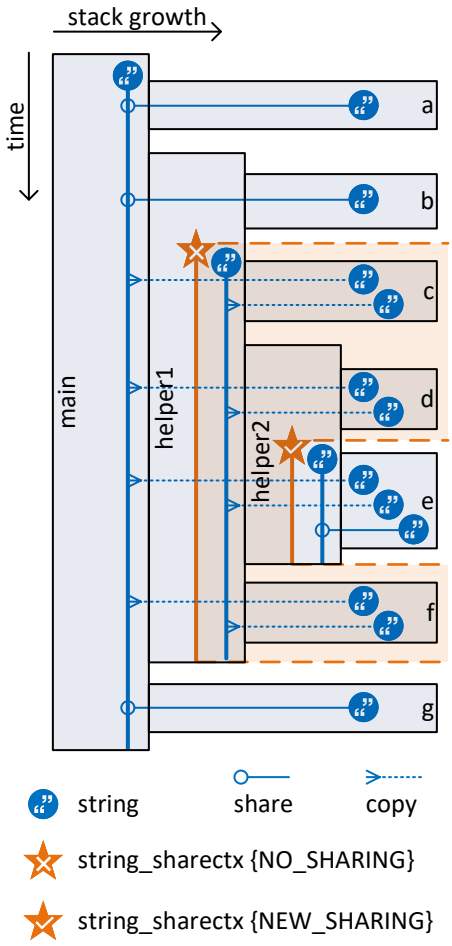


Figure 5.7: Controlling copying *vs.* sharing of strings using string_sharectx.

5.3.7 Short-String Optimization

The GNU C++ library implements a short-string ($\leq$ 16 bytes) optimization (SSO). As a string header contains a pointer to the string text, this pointer can be tagged and used to contain a short string, removing a dynamic memory allocation/deallocation.

```
class string {
    union {
        struct {                                // long string, text in heap
            size_t size;
            char * strptr;
        } lstr;
        char sstr[sizeof(lstr)];                // short string <16 characters, text in situ
    };
    // some tagging for short or long strings
};
```

However, there is now a conditional check required on the fast-path to switch between short and long string operations.

5.4 Performance Assessment

I assessed the C∀ string library's speed and memory usage against strings in the C++ STL. Overall, this analysis shows common features in both APIs come at no substantial cost in performance. Moreover, the comparison shows that C∀'s high-level string features simplify text processing because the STL requires users to think more about memory management. When the user does, and is successful, STL's performance can be very good. But if a user does not understand the consequences of the STL representation, performance becomes poor. The C∀ string lets the user work at the level of just putting the right text into the right variables, with corresponding performance degradations reduced or eliminated.

5.4.1 Methodology

These tests use a *corpus* of strings. The lengths are important, not the contents. In a result graph, a corpus's mean string-length is often the independent variable on the x-axis. When a corpus contains strings of different lengths, the lengths are drawn from a lognormal distribution. Therefore, strings much longer than the mean occur less often, and strings slightly shorter than the mean occur most often. A corpus's string sizes are one of:

Fixed-size all string lengths are of the stated size.

Varying 1 and up the string lengths are drawn from the lognormal distribution with a stated mean and all lengths occur.

Varying 16 and up string lengths are drawn from the lognormal distribution with the stated mean, but only lengths 16 and above occur; thus, the stated mean is above 16.

The special treatment of length 16 deals with the SSO in STL string, currently not implemented in C∀. A fixed-size or from-16 distribution ensures that C++'s extra-optimized cases are isolated

within, or removed from, the comparison. In all experiments that use a corpus, text is generated and loaded before the timing phase begins. To ensure comparable results, a common memory allocator is used for C∀ and C++. As in Section 4.6.1.5, p. 96, the llheap allocator is used throughout, including in the C++ trials.

The operations being measured take only dozens of nanoseconds, so a succession of many invocations is run and timed as a group. The experiments run for a fixed duration (5 seconds), as determined by re-checking clock() every 10,000 invocations, which is never more often than once per 80 ms. Timing outcomes report mean nanoseconds per invocation, which includes harness overhead and the targeted string API execution.

As discussed in Section 5.3.3, p. 125, general performance comparisons are made using C∀'s faster, low-level string API, named string_res. Section 5.3.5, p. 129 presents an operational mode where C∀ string sharing is turned off. In this mode, the C∀ string operates similarly to C++'s, by using a heap allocation for string text. Some experiments include measurements in this mode for baselining purposes, called "C++ emulation mode" or "nosharing". See Section 4.6.1.5, p. 96 for a description of the hardware environment.

5.4.2 Test: Append

These tests measure the speed of appending strings from the corpus onto a larger, growing string. They show C∀ performing comparably to C++ overall, though with penalties for simple API misuses. The basic harness is:

```
// set alarm duration
for ( ... ) {               // loop for duration
    // reset accum to empty
    for ( i; N ) {          // perform multiple appends (concatenations)
        accum += corpus[ f( i ) ];
    }
    count += N;             // count number of appends
}
```

The harness's outer loop executes for the experiment duration. The string is reset to empty before appending (see below). The inner loop builds up a growing-length string with successive appends. Each run targets a specific (mean) corpus string length and produces one data point on the result graph. Three specific comparisons are made with this harness. Each picks its own independent-variable basis of comparison. All three comparisons use the varying-from-1 corpus construction, *i.e.*, they allow the STL to show its SSO advantage.

5.4.2.1 Fresh vs Reuse in C++, Emulation Baseline

The first experiment compares C∀ with C++ in nosharing mode using malloc/free. This experiment also introduces the first C++ coding pitfall, which the next experiment shows is helped by turning on C∀ sharing. In the following, both programs start with accum empty and build it up by appending N strings (type string in C++ and the faster string_res in C∀).

```
for ( ... ) {                                   string_res accum;
    string_res accum;    // fresh               for ( ... ) {
    for ( N )                                       accum = " ";        // reuse
        accum += ...     // append                 for ( N )
}                                                       accum += ...    // append
                                                }
```

The difference is creating a new or reusing an existing string variable. The pitfall is that most programmers do not consider this difference. However, creating a new variable implies deallocating the previous string storage and allocating new empty storage. As the string grows, further deallocations/allocations are required to release the previous and extend the current string storage. So, the fresh version is constantly restarting with zero string storage, while the reuse version benefits from having its prior large storage from the last append sequence.

Figure 5.8 shows the resulting performance. The two fresh (solid lines) and the two reuse (dash lines) are identical, except for lengths $\leq$10, where the C++ SSO has a 40% average and minimally 24% advantage. The gap between the fresh and reuse lines is the removal of the dynamic memory allocates and reuse of prior storage, *e.g.*, 100M allocations for fresh *vs.* 100 allocations for reuse across all experiments. While allocation reduction is huge, data copying dominates the cost, so the lines are still reasonably close together.

5.4.2.2 C∀'s Sharing Mode

This comparison is the same as the last one, except the C∀ implementation is using sharing mode. Hence, both C∀'s fresh and reuse versions have no memory allocations, and as before, only for reuse does C++ have no memory allocations. Figure 5.9 shows the resulting performance. For fresh at append lengths 5 and above, C∀ is now closer to the C++ reuse performance, because of removing the dynamic allocations. However, for reuse, C∀ has slowed down slightly, to performance matching the new fresh version, as the two versions are now implemented virtually the same. The reason for the C∀ reuse slow-down is the overhead of managing the sharing scheme (primarily maintaining the list of handles), without gaining any benefit.

5.4.2.3 Misusing Concatenation

A further pitfall occurs writing the apparently equivalent `x = x + y` *vs.* `x += y`. For numeric types, the generated code is equivalent, giving identical performance. However, for string types there can be a significant difference. This pitfall is a particularly likely for beginners.

In earlier experiments, the choice of C∀ API among HL and LL had no impact on the functionality being tested. Here, however, the + operation, which returns its result by value, is only available in HL. The C∀ += is obtained by inlining the HL implementation of +, which is done using LL's +=, into the test harness, while omitting the HL-inherent extra dynamic allocation. The HL-upon-LL + implementation, is:

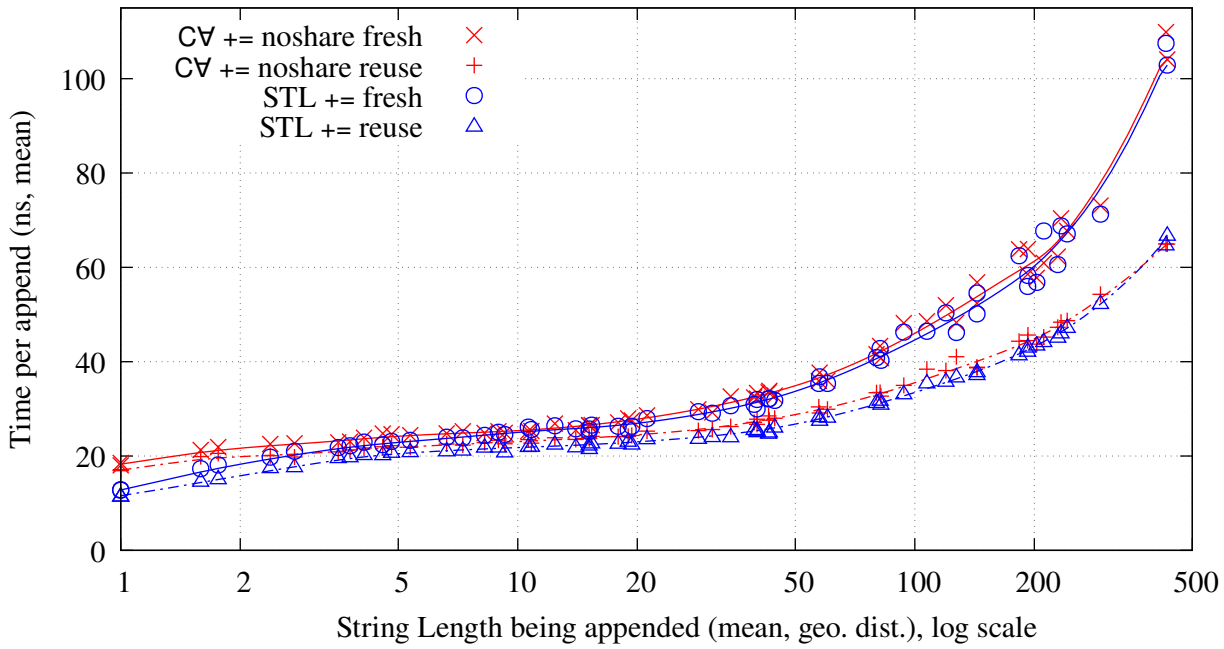


Figure 5.8: Fresh *vs.* reuse in C++, emulation baseline. Average time per iteration with one `x += y` invocation (lower is better). Comparing C∀'s STL emulation mode with STL implementations, and comparing the fresh with reused reset styles.

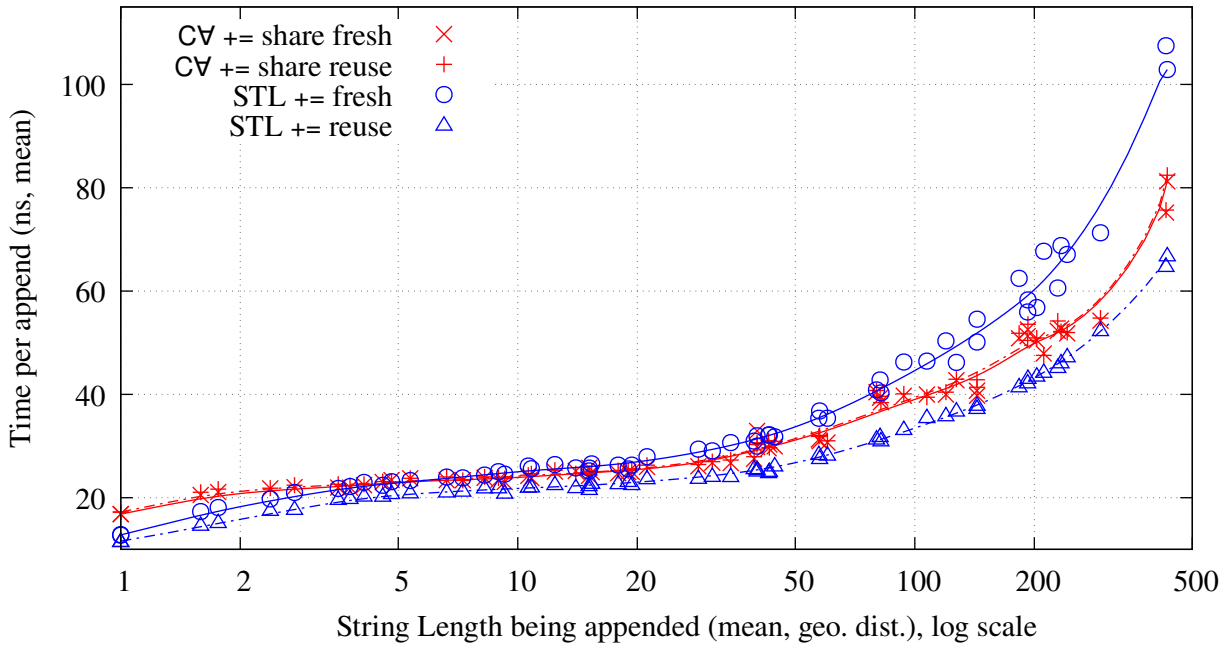


Figure 5.9: C∀ compromise for fresh *vs.* reuse. Average time per iteration with one `x += y` invocation (lower is better). Comparing C∀'s sharing mode with STL, and comparing the fresh with reused reset styles. The C++ results are repeated from Figure 5.8.

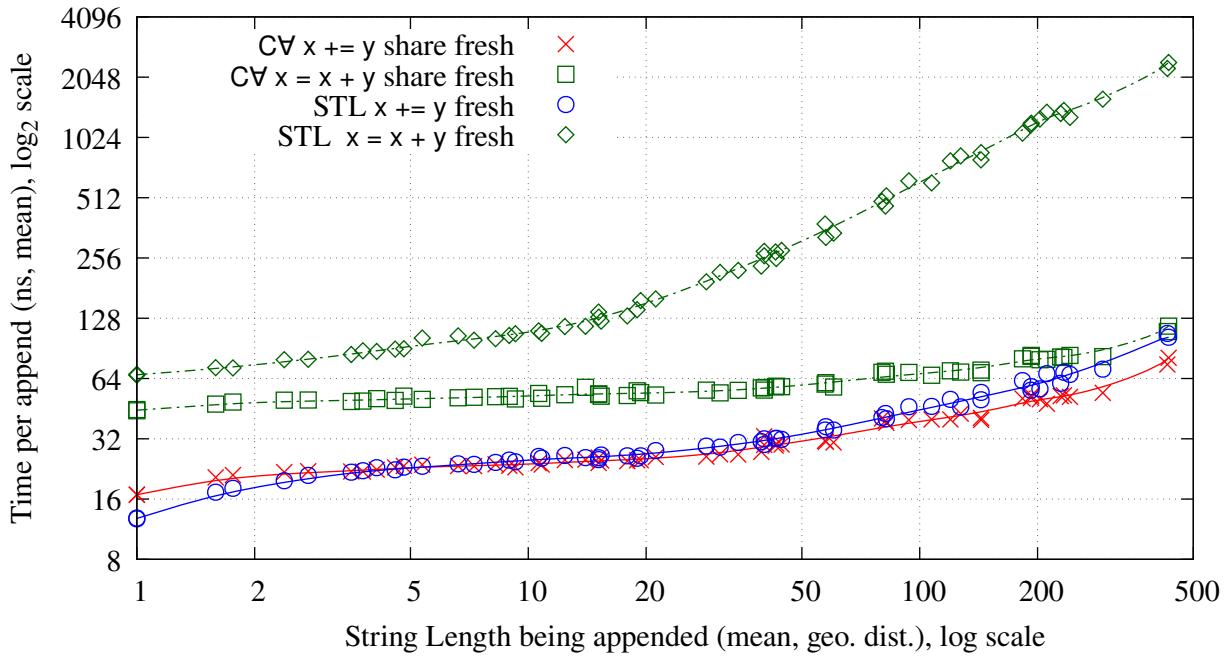


Figure 5.10: Cv's low overhead for misusing concatenation. Average time per iteration with one x += y invocation (lower is better). Comparing Cv (having implicit sharing activated) with STL, and comparing the +-then-= with the += append styles. The += results are repeated from Figure 5.9.

```
struct string {   // simple ownership
    string_res * inner;  // RAII manages malloc/free
};
void ?+=?( string & s, string s2 ) {
    (*s.inner) += (*s2.inner);
}
```

```
string ?+?( string lhs, string rhs ) {
    string ret = lhs;
    ret += rhs;
    return ret;
}
```

This + implementation is also the goal implementation of + once the HL/LL workaround is no longer needed. Inlining the induced LL steps into the test harness gives:

Goal

```
for ( ... ) {
    string accum;
    for ( ... ) {
        accum = accum + ...



    }
}
```

Measured

```
string_res pta_ll_temp;
for ( ... ) {
    string_res accum;
    for ( ... ) {
        pta_ll_temp = accum;
        pta_ll_temp += ...;
        accum = pta_ll_temp;
    }
}
```

The goal code functions today in HL but with slower performance.

Figure 5.10 gives the outcome, where the Y-axis is log scale because of the large differences. The STL's penalty is $8\times$ while Cv's is only $2\times$, averaged across the cases shown here. Moreover,

the STL's gap increases with string size, while C∀'s converges. So again, C∀ helps users who just want to treat strings as values, and not think about the resource management under the covers.

5.4.3 Test: Pass Argument

STL has a penalty for passing a string by value, which forces users to think about memory management when communicating values with a function. The key C∀ value-add is that a user can think of a string simply as a value; this test shows that C++ charges a stiff penalty for thinking this way, while C∀ does not. This test illustrates a main advantage of the C∀ sharing algorithm (in one case). It shows STL's SSO providing a successful mitigation (in the other case).

The basic operation considered is:

```
void foo( string s );
string s = "abc";
foo( s );
```

With implicit sharing active, C∀ treats this operation as normal and supported. Again, an HL-LL difference requires a mockup as LL does not directly support pass-by-value.

Goal

```
void helper( string q ) {

}
for ( ... ) { // loop for duration
    helper( corpus[ f( i ) ] );
    count += 1;
}
```

Measured

```
void helper( string_res & qref ) {
    string_res q = { qref, COPY_VALUE };
}
```

The goal (HL) version gives the modified test harness, with a single loop. Each iteration uses a corpus item as the argument to the function call. These corpus items are imported to the string heap before beginning the timed run.

Figure 5.11 shows the costs for calling a function that receives a string argument by value. STL's performance worsens uniformly as string length increases, except for short strings due to SSO, while C∀ has the same performance at all sizes. While improved, the C∀ cost to pass a string is still nontrivial. The contributor is adding and removing the callee's string handle from the global list. This cost is $1.5\times$ to $2\times$ over STL's when SSO applies, but is avoidable once C∀ realizes this optimization. At the larger sizes, the STL runs more than $3\times$ slower, because it has to allocation/deallocate storage for the parameter and copy the argument string to the parameter. If the C++ string is passed by reference, the results are better and flat across string lengths like C∀.

5.4.4 Test: Allocate

This test directly compares the allocation schemes of the C∀ string (sharing active), with the STL string. It treats the C∀ scheme as a form of garbage collection (GC), and the STL scheme as an application of malloc-free. The test shows that C∀ enables faster speed in exchange for greater memory usage.

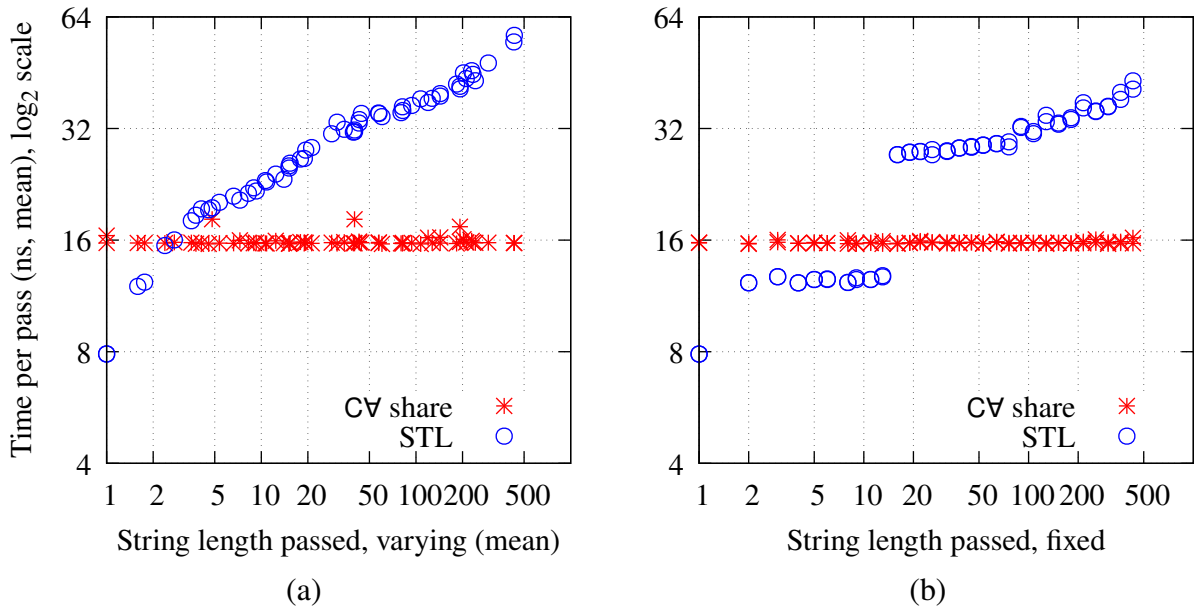


Figure 5.11: Impact of C++ small-string optimization on experimental timings. Average time per iteration (lower is better) with one call to a function that takes a by-value string argument, comparing C∀ (having implicit sharing activated) with STL. (a) With *Varying 1 and up* corpus construction, in which the STL-only benefit of SSO optimization occurs, in varying degrees, at all string sizes. (b) With *Fixed-size* corpus construction, in which this benefit applies exactly to strings with length below 16.

A precise garbage-collector, which knows all pointers and may modify them, often runs faster than malloc-free in an amortized analysis, even though it must occasionally stop to collect (latency issues). The speedup happens indirectly because GC is able to move objects. (In the case of the mini-allocator powering the C∀ string library, objects are runs of text.) Moving objects creates a large contiguous store of available memory; fresh allocations consume from this area using light-weight *bump-allocation*. A malloc-free implementation cannot move objects because the location of allocated objects is unknown; hence, free-space management is more complex managing linked structures of freed allocations and possibly coalescing freed blocks.

GC occurs lazily, so the lifetime of unreachable allocations is unknown, *e.g.*, GC may never be triggered. A garbage collector can minimize the memory footprint for unreachable allocations by collecting eagerly. However, this approach can negate GC's speed advantage. Alternatively, collecting infrequently can consume large amounts of resident and virtual memory. In contrast, a program using malloc-free tries to release an allocation as soon as it is unreachable. This approach reduces the memory footprint (depending on the allocator), but has a per allocation cost for each free. Therefore, a speed *vs.* memory tradeoff is a good basis for comparing the allocation schemes of C∀ (GC) *vs.* STL (malloc-free).

These tests manipulate a tuning knob that controls how much extra space to use. Specific values of this knob are not user-visible and are not presented in the results. Instead, its two effects (amount of space used and time per operation) are shown. The independent variable is the liveness target, which is the fraction of the text buffer in use at the end of a collection. The allocator expands its text buffer during a collection if the actual live fraction exceeds this target.

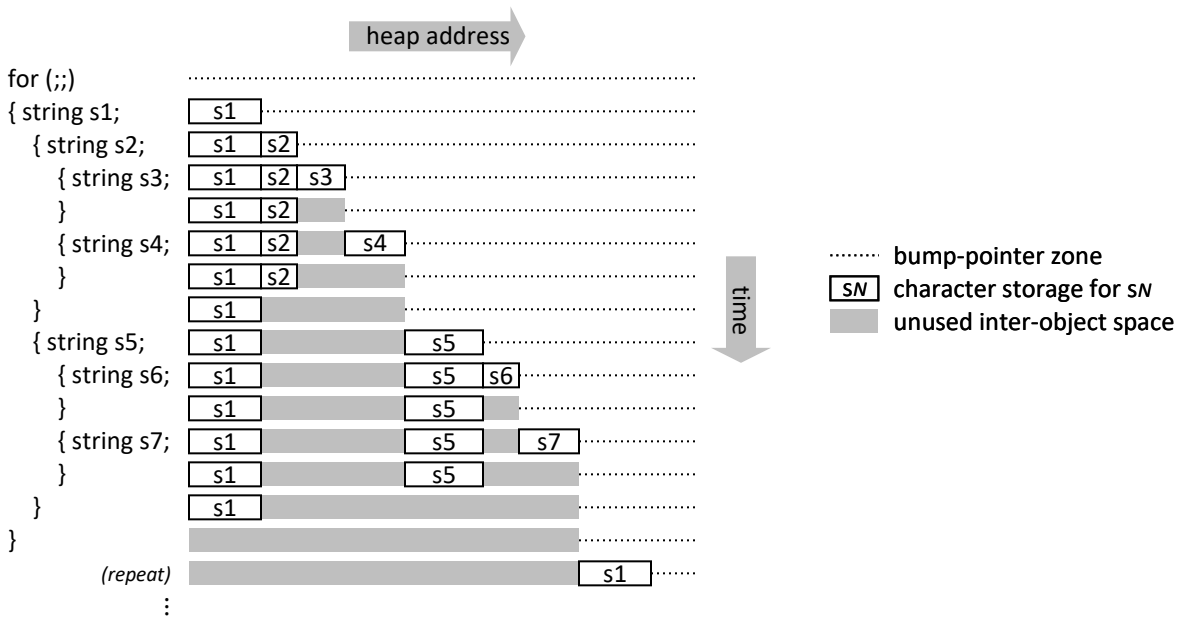


Figure 5.12: Memory-allocation test harness and its usage pattern. The pattern is shown early in a test run, when only bump-pointer allocation has occurred so far.

Figure 5.12 shows the memory-allocation pattern that the test harness drives. Note how this pattern creates few long-lived strings and many short-lived strings. To extend this pattern to the scale of a performance test, the harness is actually recursive, to achieve the nesting (three levels in the figure), and iterative, to achieve the fan-out (factor two / binary tree in the figure).

```
void f( int depth ) {
    if (depth == 0) return;
    string_res x = corpus[...]; // importing from char * here
    COUNT_ONE_OP_DONE
    for ( fanOut ) {
        // elided: terminate fixed−duration experiment
        f( depth − 1 );
    }
}
START_TIMER
f( launchDepth );
STOP_TIMER
```

The runs presented use a call depth of 1000 and a fan-out of 1.006, which means that approximately one call in 167 makes two recursive calls, while the rest make one. This sizing keeps an average of 836 strings live and keeps the amounts of consumed memory within the machine's last-level cache. The time measurement is nanoseconds per f-call, for internal or leaf strings.

Figure 5.13 shows the experimental results. In plot (a), there is one labeled curve for each of the five chosen string sizes, 20, 50, 100, 200, 500. A labeled curve shows the C∀ speed and space combinations achievable by varying the fraction-live tuning parameter. Stepping along the curve from top-left toward bottom-right means the string library has a smaller fraction of its heap remain

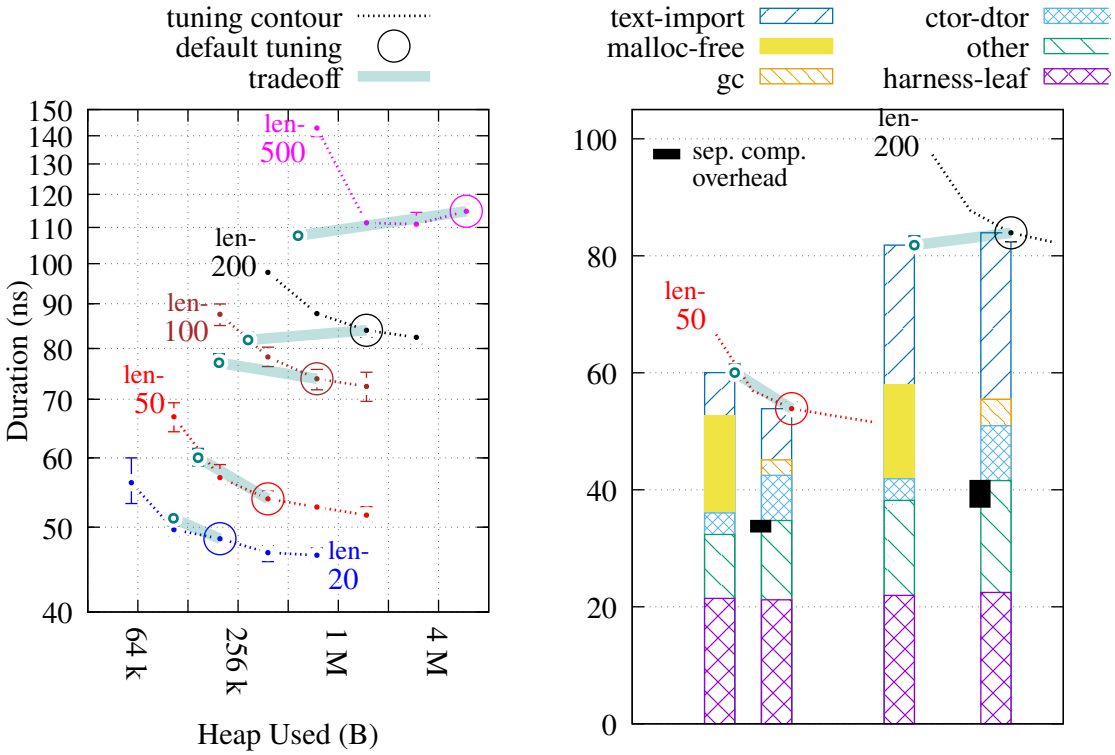


Figure 5.13: String allocation space and time performance. The experiment varies the fraction-live target, at the five string lengths shown, under *Varying 16 and up* corpus construction.

live after collection, *i.e.*, it doubles its size more often. Hence, memory approximately doubles between points. The additional memory means less time is spent collecting, so the curve shows lower (better) times per operation (y-axis). This doubling benefit diminishes, so the curves begin to flatten.

The point chosen as C∀'s default liveness threshold (20%) is marked with a circle. For most string lengths, this point occurs as the doubling benefit becomes subjectively negligible. At len-500 (top line in plot (a)), the amount of space needed to achieve 20% liveness is so much ($>$ 4M) that last-level cache misses begin occurring generating a further slowdown. This effect is an anomaly of the experimental setup trying to compare such varied sizes in one view; the len-500 default point is included only to provide a holistic picture of the STL comparisons (discussed next).

Staying within plot (a), each *tradeoff* line (green) connects the default-tuned C∀ performance point with its length-equivalent STL performance point. STL always uses less memory (occurs to the left of) than its C∀ equivalent. Ideally, C∀ results would be faster (STL occurs above it); however, this inversion only occurs for smaller string lengths. That is, the STL-to-C∀ tradeoff line goes down and to the right for small strings, but for longer strings, it goes up and to the right.

Plot (b) offers insight into why larger lengths are not currently showing the anticipated tradeoff, and suggests possible remedies. This plot breaks down the time spent, comparing STL–C∀ trade-

offs, at successful len-50 with unsuccessful len-200. Data are sourced from running the experiment under *perf*, recording samples of the call stack, and imposing a mutually-exclusive classification on these call stacks. Reading a stacked bar from the top down, *text import* captures samples where a routine like memcpy is running, so that the string library is copying characters from the corpus source into its allocated text buffer. *Malloc-free* means literally one of those routines (taking non-trivial time only for STL), while *gc* is the direct C∀ (-only) equivalent. All of the attributions so far occur while a string is live; further time spent in a string's lifecycle management functions is attributed *ctor-dtor*. Skipping to the bottom-most attribution, *harness-leaf* represents samples where the youngest live stack frame is simply the recursive helper routine of Figure 5.12, p. 138. The skipped attribution *Other* has samples that meet none of the criteria above.

Beside C∀'s principal bars, a bar for separate-compilation overhead (*sep. comp.*) shows the benefit that STL enjoys by using monolithic compilation. C∀ currently forgoes this benefit. This overhead figure is obtained by hacking a version of the C∀ string library to have a header-only implementation and measuring the resulting speed. The difference between separately compiled (normal) and header-only (hacked) versions is the reported overhead. It represents how much C∀ could speed up if it switched to a header-only implementation to match STL. The difference jumps noticeably between the different string sizes, but has little correlation with string size ($r = 0.3$). Associating this potential improvement with the *other* attribution is a stylistic choice.

Analyzing the stacked bars from the bottom up, the first two categories, *harness-leaf* and *other* both give baseline times that should not be matters of STL-C∀ difference. The *harness-leaf* category is called out as a quality check; this attribution is equal across all four setups, ruling out mysterious "everything is X% slower" effects. But with the *other* attribution, STL beats C∀ by an amount that matches the STL's benefit from avoiding separate compilation. In the *ctor-dtor* category, C∀ is spending more time than STL to inter-link its string handles. Comparing STL's *malloc-free* with C∀'s *gc*, C∀'s is considerably shorter, in spite of its copying, because *gc* only spends time on live strings, while *malloc-free* works on all strings. Most of the every-string bookkeeping occurs under STL's *malloc-free* and C∀'s *ctor-dtor*. So the critical comparison is STL's (*ctor-dtor* + *malloc-free*) *vs.* C∀'s (*ctor-dtor* + *gc*). Here, C∀ is a clear winner. Moreover, the discussion so far holds for both the successful and unsuccessful string lengths, albeit with the length-correlated duration of *gc* (due to copying) encroaching on C∀'s advantage, though not eliminating it. The total attributions so far see C∀ ahead, possibly by more than is shown, depending on how one views the separate-compilation difference. Under a C∀-generous separate-compilation stance, C∀ is equal or ahead in every important category so far.

The remaining attribution, *text import*, is thus a major reason that C∀ is currently unsuccessful at delivering improved speed at large sizes. C∀'s loss to STL on *text import* occurs somewhat at length 50, and is pronounced at length 200. If the STL monolithic compilation advantage is removed from consideration, the *text-import* difference is the only reason that C∀ is not beating STL on speed, by about 10%, across the board.

An investigation into the *text-import* difference revealed an interesting optimization opportunity. Both implementations use a memcpy operation, sourcing from the program's argv strings (corpus strings specified on the command line), targeting the string library's text buffer. In both implementations, the memcpy code is inlined. However, at runtime, the STL's version runs faster, by doing the data movement with vector instructions, while C∀'s does not. This STL optimization

```
string greeting = "Grüße, 世界! — avec café ☕";
sout | greeting;
sout | "len (bytes): " | len(greeting);

sout | "Type a word: " | nonl;
string word;  sin | word;
if ( word ) {
    sout | "Read" | len(word) | "bytes: " | word;
}

string buf = "prefix: ";
buf += "naïve résumé façade";
sout | buf;

string csv = "α,ββ,γγγ,^,Δ,≙,📐";
while ( string p; p = toksplit(csv, ",") )
    sout | "[" | p | "]" | nosep | len(p) | nonl;
sout | nl;
```

```
$ cat in.txt
Olá-мир!


$ ./a.out < in.txt
Grüße, 世界! — avec café ☕
len (bytes): 35
Type a word: Read 12 bytes: Olá-мир!
prefix: naïve résumé façade
[α]2 [ββ]4 [γγγ]6 [^]1 [Δ]2 [≙]3 [📐]4
```

Figure 5.14: C∀ program illustrating UTF-8 support. Includes literals, input, analysis, synthesis and output.

only occurs when the source and destination are aligned on a memory boundary matching with vector-data alignment (64-byte alignment). However, strings in the C∀ text buffer are only byte aligned, whereas the C++ SSO and malloced strings are 16-byte aligned, increasing the possibly of vector alignment or an optimization that ultimately results in vector operations. The C∀ implementation may benefit from such a scheme by wasting a small amount of space to position strings at a larger alignment boundary. At present, incorporating this optimization into the heap management is too disruptive. So, this discovery is left as a potential improvement.

5.5 Unicode character support

In the Linux realm, UTF-8 is the canonical format for Unicode text in files or on the wire. Figure 5.14 shows C∀ successfully handling UTF-8 in a pass-through fashion. The example first forms a UTF-8 string literal, featuring several multi-byte characters, which it prints. The len function reports on this string's byte length (35) rather than its character count (24). Next, a string is read and echoed; when its input is multi-byte, it roundtrips well. Next, a string concatenation occurs, again succeeding with a multi-byte operand. Finally, a loop analyzes a delimited string, obtaining chunks whose byte lengths vary, first because of logical character counts, then single-character encoding length.

A C∀ string API feature helps here, to prevent code pollution due to byte length *vs.* character count. In the example's final CSV stage, note that positions are not part of the API calls—there are no numeric arguments. Rather, the loop's generator, toksplit, is returning sub-strings of the original. Offering an API that avoids framing questions and answers in terms of (byte) lengths helps users avoid situations where they may conflate the two senses of length. This pattern occurs with all of C∀'s analysis functions. Along with overloads that return byte-count numbers, the API offers overloads that return substrings, thereby shifting necessary "split_pos + 1" logic into the C∀

runtime library.

C∀'s null-terminator-free representation shines in this case. Here, a range-filtered view of the original is achieved trivially. C's strtok equivalent literally overwrites the original text allocation with null terminators, destroying any subsequent ability to get *e.g.*, strlen(original).

This C∀ program derives its multi-byte support simply by avoiding a "length = character count" assumption. The equivalent program done with C printf/strtok/*etc.* also works, on the same basis. In fact, the C∀ demo relies on this C-string support frequently, with every string literal that occurs in it.

Conversely, simply avoiding the single-byte assumption does not achieve the following desires.

1. Acknowledging several meanings of "logical character," *i.e.*, that, even after accepting multi-byte grouping as important, potential ambiguity remains, concerning just how big a group should be. Unicode defines these grouping levels, from finest and most clearly relevant in general-purpose string processing (*vs.* more specialized handling like visual rendering), to coarsest and least so:
   - *code point*, the unit on which Unicode character tables are indexed, *i.e.*, the `"\u0000"` and `"\U00000000"` numbers, all of which are expressible in four bytes or less;
   - *grapheme cluster*, where several code points combine into a unit that receives one glyph, *e.g.*, the two-character option for giving accents (see #2), an emoji with modified skin colour, or several emojis combined into a family; and
   - *orthographic syllable*, a grouping scheme common to many South Asian scripts with a glyph-per-syllable basis, which traditionally do not use spaces to separate (multi-syllable) words. These scripts traditionally allow line breaks between syllables *except* when the syllables combine, according to Unicode-codified rules of regular-expression complexity. Within one of these combinations, rendering is further ligatured, but splitting the ligature would break the text's meaning. Such a loss of meaning does not occur in English ligature patterns; *e.g.*, the *ffi* in *muffin* does not preclude line breaking as *muf-fin*.

   In each variable-width encoding, the role of an individual code unit (byte for UTF-8, byte-pair for UTF-16) in forming a code point can be discerned from inspecting just that code unit in isolation. A code point's role in a grapheme cluster or orthographic syllable requires context from inspecting a bounded number of neighbouring code points. None of these divisions requires a linear scan from a starting point.
2. Recognizing that multiple distinct code point sequences can produce the same grapheme cluster, a fact significant for equality testing; *e.g.*, the French *être* is from an equivalence class that includes code point sequences
   - [ê (\u00EA)] [t] [r] [e], and
   - [e] [◌̂ (\u0302)] [t] [r] [e].
3. Offering lengths and positions in terms of logical characters, rather than bytes. So, offering operations like, "Get character at character number," beyond C∀-current "Get byte at byte number." In a variable-width encoding, indexing by character number requires a linear scan from the start, for any of the three character definitions. The earlier API quality, of delivering fragments rather than indices, suggests the hypothesis that publishing a character-based

position numbering scheme can be avoided. A hybrid scheme is conceivable, "Get character at byte number," which requires a well-defined handling for the case, "That byte number is not the start of a character."

4. Making operations that interpret individual characters work also on multi-byte characters. In the example of Figure 5.14, p. 141, note how the parsing loop triggers on the comma, a single-byte character. Naively including a multi-byte character in the search-for string does not currently work (neither in C∀, nor with C's strtok, where both treat the delimiters' "string" as programmer shorthand for a set of *byte*-characters).
5. Using byte-count semantics where warranted, *e.g.*, during replacement, for the lengths of the text originally selected to be replaced (destination) and the replace-with text (source). "Replace *e* with *ê*," is a byte-length changing operation (in UTF-8), subject to the rules that Figures 5.3, p. 121 and 5.4, p. 122 implement, even though the operation does not change the string's character-length. Accordingly, in this respect, the C∀ current implementation is using a byte-oriented sense of length and position *correctly*.
6. Protecting against creating UTF-8/16-illegal byte patterns via incomplete destination or source fragments. Note that character-based overloads offer a means of defining incomplete fragments. For one example, s[i] = '\xff' corrupts a UTF-8 string that was previously ASCII-valid. For another, s[4] = 'a' corrupts a string where x[3] is the start of a multi-byte character.
7. Extending support to UTF-16 and -32 based strings, where C∀ is currently limited to fixed single-byte formats and UTF-8. At a conceptual level, it seems desirable to define string as a UTF-8 defaulted instantiation of a polymorphic type, string_x(char8_t), allowing string_x(char16_t) to be for UTF-16. Under today's C∀ polymorphism implementation, this change would cause a severe performance penalty, turning integer CPU operations into function calls. So, the chosen design needs to accommodate a suitable workaround for this fact, perhaps including tactical measures of duplication or a decrease in separate compilation.
8. Selecting among a fixed k-byte semantics for the above operations (*e.g.*, one may want with $k = 1$ for processing Latin-1 text) and UTF(-$8k$) semantics. That is, turning off UTF-8/-16 interpretation/enforcement in user-specified cases, when a different encoding is intended. Turning off multi-byte character awareness (or setting it to a trivial mode) achieves Latin-1, while achieving Shift_JIS (among others) would require setting the multi-byte awareness to a further nontrivial mode.

Often, the C standard does not cater to these desires either, opting rather for minimalism and leaving support to third-party libraries. Standard C offers basic conversions and interpretations through a set of "mb (multibyte)" functions in <stdlib.h> [40]. For example, mblen reports on length (generalized strlen) and mbtowc converts from UTF-8 packed to UTF-32 unpacked (generalized strcpy). These functions work at the code-point level, with no awareness of grapheme clusters or orthographic syllables (minimal support for #1 and none for #2). They provide a means for writing a loop that counts logical characters encountered, but do not offer a by-character indexing abstraction, nor do they provide a range-selection type that abstracts from numeric positions (no particular value-add for #3). This set of functions does not include a generalized strtok (no support for #4). The foundational C string APIs do not help with length-changing modifications, so there is no discussion of adapting this ability for multi-bytes (#5 and #6 inapplicable). The C landscape

has the programmer pick a character (array element) type and then pick the applicable function to operate on it (strlen, wcslen, mblen). So, basic C satisfies #7 and #8 trivially, where these qualities become relevant under a more uniform abstraction (string, len)

5.6 Future Work

C∀ strings provide an interesting set of operations for manipulating sets of characters. However, experience is needed to determine which operations are superfluous and which are missing. It is hard to find a strong string API in other general-purpose programming-languages to compare with, as most have simple string APIs, especially I/O. Supporting regular-expressions, as in text-editors and specialized string languages, *e.g.*, AWK [4], Ruby [45], SNOBOL [19], is an important extension. String sharing also needs experience to determine if the rules for overlapping behaviour provide the right features for complex editing applications.

There is room for improving compatibility where C∀ string code needs to use C **char** * code. One insight is that many C∀ strings are not substrings, *i.e.*, do not stop any earlier than an originally imported run of text. Adding a C∀-redundant null terminator after these imported runs would allow for a C++-style in-place C view (c_str), *in this common case*. Maintaining a uniform API for (non-)substrings is an important C∀ goal, but the case distinction might be achievable with a runtime check of an acceptable cost. Furthermore, using the C∀ string heap as the allocated target of a copied substring, could ease the user's memory management burden, providing that a suitable integration into the RAII system is determined.

Extending C∀ strings beyond single-byte characters is also a significant upgrade required for non-Latin languages, as detailed in Section 5.5, p. 141. Working with fixed-size 16 and 32-bit characters is a more straightforward extension from 8-bit characters, but not without design choices and tradeoffs. Variable-sized UTF characters require a ground-up consideration for implementation. For example, high-performance manipulation may require a companion array of references into the variable-sized characters, or an idiom of unpacking into a fixed-width encoding may need to be encouraged or streamlined. Supporting operations directly on variable-width text offers the large space of design decisions already summarized, ranging from ergonomic unknowns like whether user-programmers can solve variable-width compatible problems without needing a character-at-position API, through quality-of-life imperatives like resisting corruption in a variable-width encoding, to points that the current implementation correctly treats as byte oriented like, "Replace *e* with *ê*," causing a byte-length change.

Finally, adding the C++ SSO trick and merging the two-part string-implementation, once the RAII problem in C∀ is fixed, will provide even better performance.

Chapter 6

Conclusion

This thesis performed a detailed examination of the three most important high-level collections in many programming languages: array, linked-list, and string. The goal of the work is to make collections easier to use, performant and safer. This goal is relevant because a subset of these three collections are used in almost every program in every programming language. Accomplishing this goal in C is difficult because these features are poorly designed. In contrast, C∀'s advanced type system and language features plus my critical design choices made it possible to provide better support with significant safety. These choices aim at enabling application code that is easier to write, understand, maintain, and significantly safer from hacker attack vectors. Finally, the performance sections for each collection demonstrated that safety does not have to reduce performance, *i.e.*, rolling-your-own collection is now an excuse not a reason.

6.1 Arrays

The key takeaway for arrays is that the type system must be extended to properly manage array bounds (dimensions) to safely pass arrays to (polymorphic) functions. By adding a special kind of template constant to C∀, [N], the type system understands array bounds and implicitly associates these bounds with array instances, statically and dynamically, throughout the programming language. Array overruns are no longer possible because all subscripting is checked, as in other modern languages. Subscript checking can be implicitly elided when the compiler is given sufficient information to determine the subscript variable is always in bounds, giving performant execution. Indirectly, simple and complex VLA's reduce heap allocation resulting in better performance (up to an order of magnitude in comparisons with std::vector) and no contention in concurrent programs. Any reduction of dynamic allocation is always a safety benefit. Finally, language subarray slicing of higher-dimensional arrays is also a powerful and safety critical feature versus programmers attempting it themselves.

6.2 Lists

The key takeaway for lists is that intrusive lists can be made easy to use, performant by eliminating implicit memory allocation, and able to simulate wrapped lists, whereas wrapped lists cannot simulate intrusive lists. As well, the C∀ list-API eliminates many frustrating and error-prone requirements with C++ STL lists, *e.g.*, a node can be returned and removed in a single operation (no 2-step

front/pop), and an arbitrary node can be added/removed without the need for an iterator. As for C++, the C∀ list API is powerful enough to reduce almost all common C list errors, substantially increasing program safety.

6.3 Strings

The key takeaway for strings is that the C∀ library provides powerful and safe block operations for manipulating strings, at a modest performance cost. Manipulating strings is always going to be expensive because of their dynamic variable sizing, as the hardware rarely has string-level operations, possibly only move and compare. The new C∀ string API is a very expressive set of safe string operations for composing, comparing, and decomposing arbitrary length strings, include complex reading and printing operations. All low-level mechanisms for working with string, in any programming language, are error-prone in correctness and safety (back to working with arrays). Providing string lengths is also a significant safety improvement, while still working with compiler null-terminate string constants (as for C++). As well, overloading C string operations to work with C∀ strings provides a safe vector for updating existing C programs manipulating strings. Finally, creating bespoke storage management for strings has the advantage of faster, more compact storage management due to string sharing, at the cost of additional external fragmentation between the string and general heap. With the large amounts of available memory, this approach is a viable tradeoff.

6.4 Future Work

All three forms of collections presented in this work are in their nascence in C∀, both in design and implementation. This work provides the foundation for future C∀ students to add more functionality along with robust and performant implementations. Completing the syntax change from array to C-style for dimensions and subscripts is an important first step to fit with C-programmer expectation resulting in faster adoption. Also, adding a panoply of additional intrusive lists, *e.g.*, stack, queue, dequeue, *etc.*, is needed to complement the list structure. And once the RAII problem in C∀ is fixed, the two-part string implementation can be merged into a single form, possibly with even better performance. I left all the infrastructure to accomplish these tasks; they just need time and effort to complete.

References

[1] Learn adacore: Arrays. https://learn.adacore.com/courses/intro-to-ada/chapters/arrays.html. 77

[2] The unicode standard, version 17.0 - core specification. https://www.unicode.org/versions/Unicode17.0.0/core-spec/, 2025. 36

[3] Defense Advanced Research Projects Agency. Eliminating memory safety vulnerabilities once and for all. https://www.darpa.mil/news-events/2024-07-31a, July 2024. 4

[4] Alfred V. Aho, Brian W. Kernighan, and Peter Weinberger. *The AWK programming language*. Addison-Wesley Longman Publishing Co., Inc., 75 Arlington Street, Boston, MA, USA, September 1987. 144

[5] David Anderson. Clockwise/spiral rule. https://c-faq.com/decl/spiral.anderson.html, May 1994. 9

[6] Gunter Blache. Handling index-out-of-bounds in safety-critical embedded C code using model-based development. *Software & Systems Modeling*, 18:1795–1805, 2019. 4

[7] Andrew Lilley Brinker. Memory safety for skeptics. *CACM*, 69(2):52–58, January 2026. 4

[8] Peter A. Buhr. *μC++ Annotated Reference Manual, Version 7.0.0*. University of Waterloo, Waterloo Ontario, Canada, August 2025. https://plg.uwaterloo.ca/~usystem/pub/uSystem/uC++.pdf. 29, 67

[9] Tom A. Cargill. Controversy: The case against multiple inheritance in C++. *Computer Systems*, 4(1):69–82, Winter 1991. 80

[10] Vint G. Cerf. ASCII format for network interchange. https://www.rfc-editor.org/info/rfc20/, October 1969. Request for Comments 20. 36

[11] C$\forall$ Features. https://plg.uwaterloo.ca/~cforall/features, 2026. 2, 39

[12] Cristina Cifuentes and Gavin Bierman. What is a secure programming language? In *3rd Summit on Advances in Programming Languages*, pages 3:1–3:12. LIPICS SNAPL, Germany, July 2019. 1

[13] Clang: a C language family frontend for LLVM. https://clang.llvm.org. 4

[14] Archibald Samuel Elliott, Andrew Ruef, Michael Hicks, and David Tarditi. Checked C: Making C safe by extension. In *2018 IEEE Cybersecurity Development (SecDev)*, pages 53–60, November 2018. 4

[15] Mehmet Emre. *Translating C to Safe Rust: Reasoning about Pointer Types and Lifetimes*. PhD thesis, UC Santa Barbara, 2022. 1

[16] Back to the building blocks: A path toward secure and measurable software. https://bidenwhitehouse.archives.gov/wp-content/uploads/2024/02/Final-ONCD-Technical-Report.pdf, 2024. 4

[17] GObject reference manual. https://developer.gnome.org/gobject/stable/, 2014. 4

[18] US Federal Government. Part II: Securing the building blocks of cyberspace. https://www.whitehouse.gov/wp-content/uploads/2024/02/Final-ONCD-Technical-Report.pdf, 2024. 4

[19] R. E. Griswold, J. F. Poage, and I. P. Polonsky. *The SNOBOL4 Programming Language*. Prentice-Hall, Englewood Cliffs, 2nd edition, 1971. 144

[20] Dan Grossman. Quantified types in an imperative language. *Transactions on Programming Languages and Systems*, 28(3):429–475, May 2006. 4

[21] HaskellWiki. Dependent type. https://wiki.haskell.org/Dependent_type, April 2021. 45

[22] Troels Henriksen and Martin Elsman. Towards size-dependent types for array programming. In *Proceedings of the 7th ACM SIGPLAN International Workshop on Libraries, Languages and Compilers for Array Programming*, ARRAY 2021, pages 1–14, New York, NY, USA, 2021. Association for Computing Machinery. 76

[23] International Business Machines. *OS and DOS PL/I Reference Manual*, 1st edition, September 1981. Manual GC26-3977-0. 112

[24] International Standard Organization, Geneva, Switzerland. *C Programming Language ISO/IEC 9899:1999(E)*, 2nd edition, December 1999. https://webstore.ansi.org/Standards/INCITS/INCITSISOIEC98991999R2005. 4, 16

[25] International Standard Organization, Geneva, Switzerland. *C Programming Language ISO/IEC 9889:2011-12*, 3rd edition, December 2011. https://www.iso.org/standard/57853.html. 4, 13, 14, 44

[26] International Standard Organization, Geneva, Switzerland. *C Programming Language ISO/IEC 9899:2018*, 4th edition, June 2018. https://www.iso.org/standard/74528.html. 4

[27] International Standard Organization, Geneva, Switzerland. *C Programming Language ISO/IEC DIS 9899:2023*, 1st edition, 2023. https://www.iso.org/obp/ui/en/#iso:std:iso-iec:9899:dis:ed-5:v1:en. 4

[28] Richard Ishida. Character encodings: Essential concepts. https://www.w3.org/International/articles/definitions-characters/, 2025. 35

[29] Itanium C++ ABI. https://itanium-cxx-abi.github.io/cxx-abi/abi.html. 126

[30] K. E. Iverson. *A Programming Language*. Wiley, New York, 1962. 112

[31] Paul Kehrer. 2021 in memory unsafety - apple's operating systems. https://langui.sh/2019/07/23/apple-memory-safety, December 2021. 1

[32] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice-Hall, Englewood Cliffs, 1st edition, 1978. 11, 35

[33] Ralph Kimball and Margy Ross. *The data warehouse toolkit : the definitive guide to dimensional modeling*. John Wiley & Sons, Indianapolis, 3rd ed edition, 2013. Fig 1.1; term as used *e.g.*, Pg. 77. 17

[34] Tamara G. Kolda and Brett W. Bader. Tensor decompositions and applications. *SIAM review*, 51(3):455–500, 2009. 16, 58

[35] Jiada Liang. Enumerated types in C∀. Master's thesis, School of Computer Science, University of Waterloo, Waterloo, Ontario, Canada, 2024. https://hdl.handle.net/10012/21067. 78

[36] std::list. https://en.cppreference.com/w/cpp/container/list. 24

[37] Mend.io (White Source Ltd.). What are the most secure programming languages? https://www.mend.io/most-secure-programming-languages, 2024. 1

[38] Matt Miller. Trends, challenges, and strategic shifts in the software vulnerability mitigation landscape. https://github.com/microsoft/MSRC-Security-Research/tree/master/presentations/2019_02_BlueHatIL, February 2019. 1

[39] Aaron Moss, Robert Schluntz, and Peter A. Buhr. C∀ : Adding modern programming language features to C. *Softw. Pract. Exper.*, 48(12):2111–2146, December 2018. 86

[40] Null-terminated multibyte strings. https://en.cppreference.com/c/string/multibyte. 143

[41] Special Interest Group on C++. Return value optimization (RVO). https://sigcpp.github.io/2020/06/08/return-value-optimization, June 2020. 126

[42] Adam Paszke, Daniel D. Johnson, David Duvenaud, Dimitrios Vytiniotis, Alexey Radul, Matthew J. Johnson, Jonathan Ragan-Kelley, and Dougal Maclaurin. Getting to the point. index sets and parallelism-preserving autodiff for pointful array programming. *Proc. ACM Program. Lang.*, 5(ICFP), aug 2021. 77

[43] queue(7) – linux manual page. https://man7.org/linux/man-pages/man3/queue.3.html. 24

[44] Dennis M. Ritchie. The development of the C language. *SIGPLAN Not.*, 28(3):201–208, March 1993. 9

[45] Ruby. *Ruby Documentation, Release 2.6.0*. Python Software Foundation, https://www.ruby-lang.org/en/documentation, 2018. 144

[46] Andrew Ruef, Leonidas Lampropoulos, Ian Sweet, David Tarditi, and Michael Hicks. Achieving safety incrementally with Checked C. In Flemming Nielson and David Sands, editors, *Principles of Security and Trust*, pages 76–98, Cham, 2019. Springer International Publishing. 4

[47] Robert Schluntz. Resource management and tuples in C∀. Master's thesis, School of Computer Science, University of Waterloo, Waterloo, Ontario, Canada, 2017. https://hdl.handle.net/10012/11830. 125

[48] Jim Shore. Fail fast. *IEEE Softw.*, 21(5):21–25, September 2004. 30

[49] Geoffrey Smith and Dennis Volpano. A sound polymorphic type system for a dialect of C. *Science of computer programming*, 32(1-3):49–72, 1998. 4

[50] Bjarne Stroustrup. Multiple inheritance for C++. *Computer Systems*, 2(4):367–395, Fall 1989. 80

[51] Bjarne Stroustrup. *The Design and Evolution of C++*. Addison-Wesley, Boston, 1994. 125

[52] Herb Sutter. *Exceptional C++: 47 Engineering Puzzles, Programming Problems, and Solutions*. Addison-Wesley Longman Publishing Co., Inc., 75 Arlington Street, Boston, MA, USA, June 1999. 87

[53] Ken Thompson. A new C compiler. In *Proceedings of the Summer 1990 UKUUG Conference*, pages 41–51, 1990. https://doc.cat-v.org/bell_labs/new_c_compilers/new_c_compiler.pdf. 79

[54] Unicode version character counts. https://www.unicode.org/versions/stats/charcountv17_0.html, 2026. 36

[55] Paul C. van Oorschot. Memory errors and memory safety: C as a case study. *IEEE Security & Privacy*, 21(2):70–76, 2023. https://ieeexplore.ieee.org/document/10102611. 1

[56] Ken Whistler and Asmus Freytag. Unicode technical report #17: Unicode character encoding model. https://www.unicode.org/reports/tr17/tr17-9.html, 2022. 35

[57] David H. White, Thomas Rupprecht, and Gerald Lüttgen. Dsi: an evidence-based approach to identify dynamic data structures in c programs. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ISSTA 2016, pages 259–269, New York, NY, USA, 2016. Association for Computing Machinery. 2

[58] Mubeen Zulfiqar. High-performance concurrent memory allocation. Master's thesis, School of Comp. Sc., Univ. of Waterloo, Waterloo, Ontario, Canada, 2022. https://hdl.handle.net/10012/18329. 96