ORIGINAL ARTICLE

Using Move Semantics in C++ to Minimize Aliasing and Simplify Reasoning

Alan Weide[1] | **Paolo A. G. Sivilotti**[1] | **Murali Sitaraman**[2]

[1]Department of Computer Science and Engineering, The Ohio State University, Columbus, OH, USA

[2]School of Computing, Clemson University, Clemson, SC, USA

Correspondence
Alan Weide, Department of Computer Science and Engineering, The Ohio State University, Columbus, OH, USA
Email: weide.3@osu.edu

Funding information

Most modern programming languages rely on pointer and reference copying for efficient data movement. When references to mutable objects are copied, aliases are introduced, often complicating formal and informal behavioral reasoning. While some aliasing (and related complexity in reasoning) is unavoidable in software development, this paper explains how aliasing, and its impact on reasoning, can be minimized for practical software development by leveraging the relatively recent introduction of *move semantics* in C++ (as of C++ 11), simultaneously maintaining important performance properties of traditional programming. Our central contribution lies in a carefully-developed discipline for programming based on move semantics to avoid most routine introduction of aliasing in programming, thereby leading to simpler reasoning about the behavior of software. The discipline requires attention to interface and implementation development. The ideas are illustrated using components that we have built with and without the use of unique pointers.

KEYWORDS
aliasing, C++, move semantics, reasoning, reuse

1 | INTRODUCTION

A central reason for the introduction of objects in modern programming languages is to facilitate abstraction and reuse, which in turn make it easier to develop high-quality software. These benefits, however, are compromised when object

references are aliased [1, 2, 3, 4, 5, 6]. In the presence of aliasing (and mutable objects), references cannot be assumed to be independent. That is, the modification of an object's value affects the value of that object as seen through aliased references—even though those other references are not mentioned in the code that changes the object's value. As a result, aliasing complicates reasoning and can undermine both abstraction and reuse.

1.1 | Aliasing and Reasoning

A language with explicit references gives rise to aliases, either through assignment statements or parameter passing. Aliases, however, create significant challenges for reasoning by undermining modularity and by forcing memory addresses to be part of the state of the program [7]. Because of *potential* aliasing, verification in popular languages such as Java requires establishing the frame property (that objects not mentioned do not change) for practically every method call. Some machinery such as separation logic [8, 3] or equivalent is absolutely necessary to ease this process [9]. When there is potential for aliasing, modular reasoning of one component at a time becomes difficult because it is not possible to reason about variables using their abstract values in specifications and instead their data representations become relevant, which breaks abstraction [10]. Though there has been progress, the automation of software verification remains a challenge when code involves objects and aliased references as discussed in Section 2.

Reasoning, whether formal or informal, is easier when a programming language has rich and clean semantics [11]. *Rich semantics* means that an object can be viewed abstractly as a value in a suitable mathematical domain (as opposed to a memory address). *Clean semantics* means that the effects of an operation are restricted to its explicit parameters. Supporting rich and clean semantics efficiently requires language support for an alternative to both reference assignment (which creates aliases) and deep-copying assignment (which is slow). It also requires careful component interface and implementation design. The benefit of such a system is that statements that do not share variables are necessarily independent. This benefit also applies to parallel programs, where parallel threads with distinct variables are necessarily non-interfering, greatly simplifying verification.

Although making deep copies of data representations of objects instead of copying object references (e.g., in assignments and parameter passing) eliminates aliasing, it comes with a prohibitive performance penalty. Therefore, several research efforts have focused on a range of solutions to eliminate or minimize aliasing, without a need to make deep copies. These efforts include the use of unique references, ideas of ownership, and swapping, among others[10, 4, 12]. A recent paper in *The Art, Science, and Engineering of Programming* contains an excellent motivation for minimizing aliasing and, indirectly, for simplified reasoning [13]. The solution proposed in that paper is the design of a new language ParaSail that, among other things, includes operators for moving and swapping as alternatives to assignments.

Unlike several of the prior efforts in this realm that involve new language design and features, this paper proposes a discipline within a practical, applied programming language, C++, with the aim of making the results broadly applicable immediately to the practicing community. The discipline leverages existing language features rather than novel extensions. The goal of this discipline is to support rich and clean reasoning by eliminating aliases where possible (and curtailing them where necessary), without compromising performance.

1.2 | A Discipline Based on Move Semantics

The motivation for adopting software disciplines in general is well-understood: programming without a strict software discipline produces code that is hard to read, even harder to understand, and nearly impossible to prove correct (if in fact it is correct). Following even the simplest of software disciplines—naming conventions, formatting conventions,

etc.—dramatically improves the readability and understandability of a program. More sophisticated disciplines, however, are needed for substantial progress towards easing (automated) reasoning. For this reason, many idioms, design patterns, style guides, and language mechanisms have been devised that attempt to improve this facet of software engineering. A common theme among many of these disciplines is *alias control*, in which aliased references are advertised, made immutable, or eliminated altogether. The discipline here shares similar objectives, but the approach relies on existing language mechanisms in a language widely used in practice, rather than introducing new capabilities to a language or starting from scratch with an entirely new language.

Since C++ 11, the C++ specification has included provisions for changing the way value-passing works in certain situations by introducing the concept of *move semantics*. A primary motivation for adopting move semantics is performance improvement related to how temporary variables such as return values and expression values are treated by the compiler. However, this relatively new addition can also be leveraged for robust software development to avoid most routine aliasing without compromising performance and without inventing new language additions. When move semantics are used, some design and specification changes must be made to data abstractions to make it explicit that some operations might not behave as expected by C++ programmers who are not used to move semantics.

1.3 | Outline

Move semantics in C++ are discussed in detail in Section 2, and additional benefits beyond performance are enumerated. The Clean++ discipline is discussed in Sections 3–6 followed by a brief discussion of our experience evaluating Clean++ in Section 7. Directions for further exploration of Clean++ are presented in Section 9 and work related to this research is discussed in Section 8. Finally, Section 10 contains our conclusions.

2 | MOVE SEMANTICS IN C++

Introduced in the C++ 11 specification, *move semantics* is a mechanism designed to improve the performance of data movement [14]. It is an efficient alternative to both traditional pass-by-value and pass-by-reference. Its efficiency stems from avoiding a deep copy by moving a value from one variable to another, leaving the old variable in an undefined state.

In C++, an expression appearing on the right-hand side of an assignment operation is one of two kinds: an *l-value* or an *r-value*. An l-value represents something occupying an identifiable location in memory and is therefore suitable for the left-hand side of an assignment, for example a variable (`x`), a pointer dereference (`*p`), or a function returning a reference (`a[3]`). An r-value, on the other hand, represents the temporary result of an expression evaluation and is not suitable for the left-hand side of an assignment, for example `new T()`, a literal, or an address-of operation (`&x`). Beginning with C++ 11, r-values are further divided into *r-value references* and *const r-values*: the former may be modified while the latter may not.

What use is it to modify an r-value, given that r-values are temporary values poised to go out of scope very soon? A key observation that led to the inclusion of move semantics is that in order to implement "moving assignment", the right-hand side might have to be modified (e.g., by being set to `nullptr` or an initial value for its type). The idea behind moving assignment is that precisely *because* the right-hand side is poised to go out of scope, the left-hand side does not need to make a copy. Rather, it is enough to merely steal ownership of the data from the right-hand side.

Move semantics offer clear performance benefits. When a parameter to a method is passed by value, or when a variable is assigned with the assignment operator, the *copy constructor* or *copy assignment operator* is called (which

might make a deep copy). For large or complex data types, copying can be expensive or difficult to implement, or both. If the value of an actual argument to a method call is not needed after the call completes, then parameter passing can be made far more efficient by *moving* the value of the actual argument to the formal parameter, leaving the actual argument in an easily constructed (or undefined) state. In Appendix **??**, a sample program is shown in which copy semantics leads to the creation of several unnecessary copies of a resource; employing move semantics eliminates this copying and results in a single allocation of the resource, whose value is then moved between variables, parameters, and return values.

2.1 | Reasoning Benefits of Move Semantics

In typical C++ programs, there are often many pointers and references. Sometimes, the use of pointers stems from the implementation of a linked data structure with unavoidable aliases, such as a directed graph. Often, however, aliases arise from efficiency concerns as they provide a mechanism for constant-time and constant-space assignment and parameter passing. For example, consider the list component from the standard template library. This component's insert method creates a copy of each inserted item. To avoid expensive copying, a list could be declared to hold pointers (`T*`) rather than the items themselves (`T`).

Unfortunately, the use of pointers and reference semantics creates significant challenges for reasoning about the behavior of code. At the root of these challenges is the fact that aliases permit a program fragment to affect variables that are not explicitly mentioned by that fragment (violating the clean semantics property). Put another way, understanding the effect of a program statement requires, in the worse case, a whole-program analysis of pointer variables and the memory addresses they contain.

Consider the aliasing illustrated in the program below.

```
int main(int argc, const char* argv[])
{
  int* a = new int[3];
  for (int i = 0; i < 3; i++) { a[i] = i+1; }
5 int* b = a;
  b[1] = 4; // modify array b
  printf("b␣=␣{%d,␣%d,␣%d}\n", b[0], b[1], b[2]);
  printf("a␣=␣{%d,␣%d,␣%d}\n", a[0], a[1], a[2]);
  return 0;
10 }
```

The program prints the same value for both `a` and `b`: `{1, 4, 3}`, even though `a` is not mentioned anywhere in the statement on line 6 that modifies `b`. Obviously this simple example is trivial to reason about, but understanding even slightly more complicated programs in a systematic, modular, or automated way is intractable in the general case (because the code that modifies `b` might be hidden inside the body of a function or method call replacing line 6).

2.1.1 | Memory Management

One specific manifestation of these challenges is the difficulty of memory management. Properly balancing memory allocation and deallocation is notoriously hard, as evidenced by the ubiquity of errors related to memory leaks, dangling pointers, and null dereferences (and by the existence of garbage collectors). Using move semantics and programming according to a strict discipline with can eliminate many of these errors. A relatively new programming language called

Rust [15] aims to tackle this class of errors by employing moving semantics by default and providing a strict set of rules whenever data is shared between variables. It is discussed in relative detail in Section 8.1.

To eliminate aliasing in the example above, we could encapsulate an array inside a class and override the constructor and assignment operator for that class, allowing only the moving versions of them. The modified code would look like the following listing.

```
class MoveArrInt
{
private:
  int m_a[];
public:
  MoveArrInt(const MoveArrInt& m_arr) = delete;
  MoveArrInt(MoveArrInt&& m_arr)
  {
    m_a = m_arr.m_a;
    m_arr.m_a = new int[0];
  }
  MoveArrInt(int*&& arr)
  {
    m_a = arr;
    arr = new int[0];
  }

  MoveArrInt& operator=(const MoveArrInt& m_arr) = delete;
  MoveArrInt& operator=(MoveArrInt&& m_arr)
  {
    if (&m_arr == this)
    {
      return *this;
    }
    delete[] m_a;
    m_a = m_arr.m_a;
    m_arr.m_a = new int[0];
    return *this;
  }

  int& operator[](std::size_t idx)
  {
    return m_a[idx];
  }
};

int main(int argc, const char * argv[])
{
  MoveArrInt a(new int[3]);
  for (int i = 0; i < 3; i++) { a[i] = i+1; }
  MoveArrInt b = std::move(a);
```

```
    b[1] = 4; // modify array b
    printf("b␣=␣{%d,␣%d,␣%d}\n", b[0], b[1], b[2]);
    printf("a␣=␣{%d,␣%d,␣%d}\n", a[0], a[1], a[2]);
45  return 0;
}
```

The `MoveArrInt` class deletes the copy constructor and copy assignment operator, opting instead to only allow construction and assignment when the argument or right-hand side is an r-value reference. The result, as seen in the `main` method, is that when constructing or assigning a `MoveArrInt` from another one, the client must enclose the argument in a call to `std::move`. Introduced in C++ 11 along with move semantics, the `std::move` operation does nothing but convert its argument to an r-value reference, effectively marking it as unowned (and hence modifiable). Because the C++ language specification does not define the value of a variable after it is moved, a moved variable should never be used after such a call until it is assigned a new value. Indeed, printing the value of `b` on line 43 displays `{1, 4, 3}` as before, but printing the value of `a` on line 44 displays garbage values.

In the modified program, there is never more than one variable that "owns" any one chunk of memory. Aliasing is eliminated, and it becomes possible to reason locally about the program: the value of `a` is not changed by the statement on line 42, nor can *any* statement not explicitly mentioning `a` change its value.

This design pattern can be extended to include a wide range of data types, including linked data types and others whose values are typically allocated and managed on the heap. By implementing move constructors and move assignment operators and also deleting their copying counterparts, aliases can be prevented while the performance benefits afforded by reference semantics are be preserved.

3 | Clean++: A DISCIPLINE FOR SOFTWARE ENGINEERING IN C++

The use of move semantics makes it possible to write clean, modular code that is easy to understand and reason about. A discipline based on move semantics would, ideally, encourage abstraction and understandability, and would guarantee the soundness of local reasoning.

In this section, we introduce a discipline called Clean++ for C++ programming based on these goals and others. Specifically, the development of Clean++ was directed by several broad goals:

- *Soundness of Local Reasoning*. A Clean++ program should exhibit behavior that is immediately apparent from locally reasoning about the code: e.g., reasoning about a function call should not require knowing the implementation details of that function. Ensuring the soundness of local reasoning also includes minimizing aliases.
- *Preservation of Good Object-Oriented Programming Practices*. A programmer writing in the Clean++ discipline should be guided by the rules and structures of the discipline to write code that makes judicious use of data abstractions to maximize modularity.
- *Familiarity*. A Clean++ program should be familiar to a C++ programmer. It should "look like" C++ and the behavior of the program should not be unexpected to an experienced C++ programmer.

In summary, the Clean++ discipline has eight rules.

Mutability

The Clean++ discipline has been developed with a specific focus on *mutable* data because immutable data preempts the reasoning problems posed by aliased references. Immutable types in Clean++ have no marked differences compared to their counterparts in idiomatic C++, so they are ignored in this discussion.

3.1 | Soundness of Local Reasoning

Local reasoning is a technique by which automated verifiers are able to tractably verify the correctness of relatively large programs. There are many language features of mainstream programming languages that complicate local reasoning—and, in some cases, break it entirely.

Perhaps the leading cause for unsound local reasoning is the prevalence of aliased references. When two variables refer to the same piece of memory, they are called *aliases*, and the modification of the data through one of the variables will change the value of the other (at least, most people would say so). It is thus desirable for a programming discipline that wishes to maintain the soundness of local reasoning to preclude aliases where possible and advertise or encapsulate them where necessary.

There are several options for maintaining the soundness of local reasoning in Clean++. The first is to prefer statically allocated "stack" variables, which by default are *copied* on assignment. Of course, the performance of stack variables is prohibitive in many cases so the possiblity of dynamically-allocated objects must be accounted for.

Experience has shown that it is possible to develop real-world software with dynamically-allocated objects that has no aliases whatsoever [16], and design patterns that support such components are preferred when applicable. A good way to prevent aliases is to use types with appropriate move constructors and move assignment operators, and to use them everywhere. An example of such a type from the standard template library is `std::unique_ptr`, a "smart pointer" that expresses singular ownership of its contents. Any assignment of a `std::unique_ptr` must be enclosed in a call to `std::move`, which relinquishes ownership. A program in which all raw pointers are replaced with unique pointers is a program in which there are no aliases.

There are, however, situations that necessitate data sharing of some form—usually aliasing. In such cases, a Clean++ program encourages the use of the `std::shared_ptr` type, which *advertises* the fact that a variable might have an alias. Put another way, in a traditional C++ program, the default for any reference is that it may be aliased. To prevent such aliasing, the programmer must do something special such as using `std:unique_ptr`. On the other hand, in a program that follows the Clean++ discipline, the default for any reference is that it may *not* be aliased and the programmer must explicitly allow such aliasing, if needed, with `std::shared_ptr`. The situations in which a shared pointer is necessary are rare and include the implementation of cyclic data structures.

Clean++ Rule 1

All pointers are instances of `std::unique_ptr`. In the rare situations when data sharing is absolutely necessary, `std::shared_ptr` is used.

Another source of unsoundness in local reasoning is null pointers. If a function takes a parameter that might be null, and it attempts to dereference a null pointer, this is a failure of local reasoning. The most obvious way to avoid null dereferences is to prevent null pointers entirely. Experience has shown that eliminating null pointers entirely is, in general, possible, but it leads to code that might be unfamiliar to C++ programmers used to dealing with nullable pointers.

In Clean++, pointers are initialized at the point of their declaration to refer to a *default object of the appropriate*

type. This leads to some important reasoning properties, though it is not always feasible for performance reasons, especially in low-level implementations of Clean++ components (*i.e.*, those that use something besides other Clean++ components), and for that reason Clean++ permits null pointers in low-level implementations provided they *are never leaked to client code*. Eliminating null references eliminates a significant category of run-time errors, making programs written in Clean++ safer by default.

> **Clean++ Rule 2**
>
> If a null pointer is used in the implementation of a Clean++ component, it may never be leaked to client code.

3.1.1 | Parameter Passing and Return Types

Every parameter to a method in a Clean++ program should be an rvalue reference. Although this rule causes Clean++ to deviate somewhat from idiomatic C++ more than other rules, the benefits of doing so far outweigh the familiarity concerns. First, rules regarding ownership force Clean++ to do something different in certain situations where, for example, several parameters to a method are to have their values changed. One alternative to solve this problem is to permit pass-by-reference, which is avoided because it creates aliases.

> **Clean++ Rule 3**
>
> Every method parameter is passed as an rvalue reference.

Since all of the parameters to a method are rvalue references (that is, they are *moved* into the method rather than copied), the arguments lose their values. Sometimes, however, keeping the argument values around is necessary for one reason or another—normally performance. Therefore, rule 3 has a closely-related counterpart, rule 4: `std::tuple` is the default return type of methods in Clean++. Each method should return both any newly-created objects and the updated values of all arguments as components of a tuple. This idiom allows Clean++ to simulate "pass-by-swap" as is implemented in verification- and reasoning-focused languages such as RESOLVE [17]. A secondary positive side effect of this rule is that is easy to statically translate more traditional-looking method headers and calls into Clean++-conformant headers and calls.

> **Clean++ Rule 4**
>
> Every method has a return type of `std::tuple`, in which the first component(s) are objects created by the method, if any, and the rest are the arguments in left-to-right order.

Exception: Member Functions

Member functions in C++ always pass their receiver by reference,[1]and because member functions are imperative to producing good C++ code, this limitation on member functions produces exceptions to the rules above: the receiver to a member function can, indeed, be passed by reference (and its value might be changed).

[1] It is possible in C++ to pass a receiver as an rvalue reference with a ref-qualified member function, but the complications that arise when doing so in a way that is compatible with the rest of Clean++ are overwhelming and therefore ref-qualified functions are not recommended in Clean++.

Exception: Single or No Return Values

Sometimes, a method will be simple enough that using a tuple as described above would involve a tuple with a single component (or be empty). In that case, of course, the method need not wrap the return value in a tuple. Situations in which this exception applies include methods for which every argument is intended to have its ownership relinquished by the client, or in which there is only one parameter (excluding the distinguished parameter). Examples of this exception in practice appear in the Clean++ Stack component below in section 5.2: neither the Push nor Pop method returns a tuple.

3.2 | Support for Good Object-Oriented Programming Practices

In Clean++, a programmer is guided, by the encapsulation restrictions on aliases and null references, to implement software components with many layers of data abstraction. Doing so allows her to have total control over aliases and to keep them confined to a single abstraction layer. The principle of alias control is, as noted above, one of the cornerstones of the Clean++ discipline. A consequence of this pattern is the following rule, rule 5.

> **Clean++ Rule 5**
>
> No method introduces an alias that is visible to the client.

A key feature of software components in Clean++ is that they implement an efficient default no-argument initializer that produces a coherent value for the type. Doing so ensures that whenever a new variable is created, the guarantee can be made that a client never sees a null pointer.[2]

> **Clean++ Rule 6**
>
> Each component implements a no-argument initializer, which can be made efficient.

The final responsibility placed on implementers of software components in Clean++ is that every component must have an efficient and "correct" implementation of `std::move`. Doing so allows a client to leverage built-in C++ move semantics to maintain efficiency without relying on pointer semantics. "Correct" in this case imposese a stronger requirement than the C++ specification. In particular, a moved object in Clean++ must be left in a *consistent initial state* for that type, as if the no-argument initializer had been called. (By contrast, the C++ language specification places no restrictions on the resulting value of a moved object).

> **Clean++ Rule 7**
>
> Each component deletes its copy constructor and copy assignment operator. Instead, it implements a move constructor and move assignment operator.

As a convention, software components and their clients written with the Clean++ discipline are declared in the `cleanpp` namespace. Every Clean++ software component extends `cleanpp::base`, which is analogous to Java's `Object` (the superclass of all Java types) but limited to types in the `cleanpp` namespace.

LISTING 3.1 The `cleanpp::base` abstract class.

```
namespace cleanpp {
```

[2] Some components, such as an Array, might use lazy initialization for performance reasons. Null pointers in such situations are not a problem because their existence is totally hidden from the client (see rule 2).

```
  class clean_base {
  public:
5   clean_base() = default;
    virtual ~clean_base() = default;
    virtual void clear() = 0;
  };
}
```

The `cleanpp::base` class defines one pure virtual method, `clear()`. The purpose of this method is to provide an efficient way to set an object's abstract value to an initial value for its type—that is, one produced by the zero-argument initializer (required by the Clean++ discipline, though its presence is not compiler-enforced). Also included are methods to facilitate the intuitive use of Clean++ objects with output streams; the `print` method should be overriden by a subclass for nicer output; it serves a similar purpose to the Java `toString()` method in the `Object` class.

Clean++ Rule 8

Each component extends the `cleanpp:base` abstract class, directly or indirectly.

3.3 | Familiarity

One way in which familiarity is achieved in Clean++ is the manner in which language constructs related to move semantics (eg, `std::move`) are encapsulated within class implementations as far as possible. This encapsulation means that client code does not include extra syntactic clutter that may be less familiar to a C++ programmer. In other words, it is possible for both novice and expert C++ programmers to adopt Clean++. The syntax is simple enough for the former, and familiar enough for the latter. These qualities hold despite a software component built in Clean++ having a somewhat different structure from the usual C++ style.

3.4 | Additional Considerations Going Beyond the Discipline

The following considerations are not necessarily a novel contribution of this paper, rather they are widely accepted principles for writing great software. They are discussed here only as examples of well-known software engineering principles that need not be cast aside to realize the reasoning benefits of writing software in the Clean++ discipline.

3.4.1 | Abstraction and Reasoning

Components in Clean++, by their modular nature, typically provide a clear distinction between the abstract value and concrete value of a variable. The benefits are similar to the use of "model" variables in [18]. A consequence of using many levels of abstraction to write software in Clean++ is that often the abstract value of a given type will be substantially different from the concrete value for that type. For this reason, most components in Clean++ do not use public fields in classes and instead opt for well-named methods and functions which reflect the *abstract* value of the type and not the *concrete* value. However, this tends to make Clean++ components less familiar to C++ programmers, and making use of public fields does not necessarily complicate reasoning. It is therefore permissible to use public fields in Clean++ components, though their use should be considered carefully.

3.4.2 | Observability and Controllability

Two design principles in identifying suitable operations for interfaces in Clean++ are *observability* and *controllability*, as motivated by Weide *et al.* to guide the design of abstract data types (ADTs) [19]. The principle of observability states that a client of a component should be able to distinguish between any two unequal values in the ADT's state space using some combination of operations on the ADT. Controllability, on the other hand, says that every value in the ADT's state space is reachable through some combination of operations of the ADT. An approach to designing software components (ADTs) using these principles as a guide tends to produce interfaces which are, in some sense, "sufficient". It is for that reason that Clean++ interfaces should be designed with these principles in mind.

4 | PROTOTYPICAL Clean++ COMPONENT STRUCTURE

The implementations of Clean++ components discussed in this paper are only one part (albeit the most complex part) of a Clean++ software component. In most cases, classes of the kind discussed here are not directly used by a client of a component; instead the client declares variables of "flex types" that are wrappers around the implementation types. A flex type in Clean++ has several properties:

1. It is essentially a wrapper around a `std::unique_ptr` to an object of some implementation class.
2. It provides a default implementation type for that object—that is, a client of a Clean++ component with a flex type need not have any information about the implementation(s).
3. It has a class hierarchy that mirrors the structure of the class hierarchy for the implementation types (e.g., if the implementation includes both a kernel and secondary interface, so does the flex type).
4. Every method in the implementation type has a sibling method in the flex type, the implementation of which is simply a redirection to the same method in the implementation.[3]

The flex type pattern closely resembles the "bridge" design pattern, but has several key differences. First, it is not intended to decouple the implementation from the abstraction—such decoupling is done via C++ abstract classes on the implementation side. As a consequence, the abstraction (*i.e.*, the flex type) is a concrete class, not an abstract class. Finally, the flex type has an interface that is always *identical* to the interface of the implementation. Of course, the flex type pattern does enjoy several of the benefits of the bridge pattern, most notably the ability to change the implementation at run-time, over the lifetime of a variable.

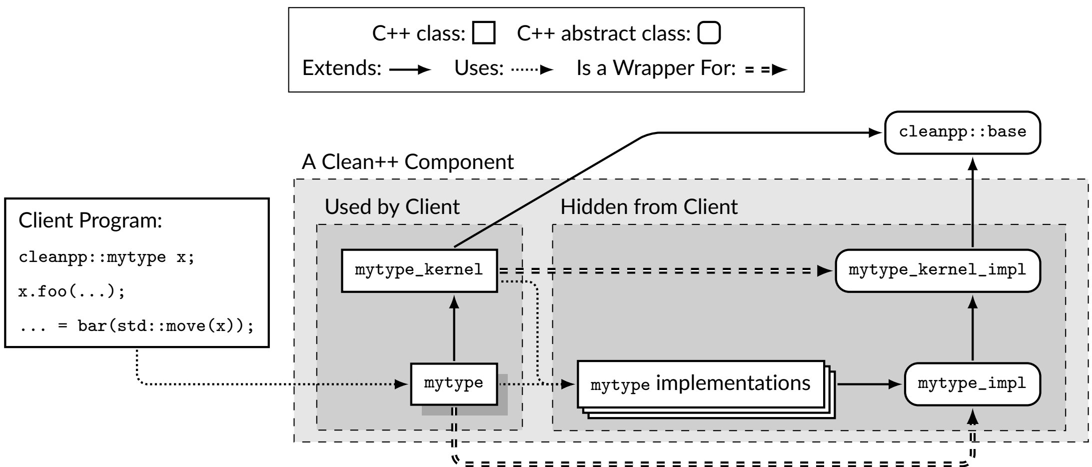
A Clean++ component family has a structure similar to figure 1. Because the flex types are the types that a client will use, Clean++ naming convention dictates that they be named with the canonical name of the component, and that the implementation type be suffixed with "`_impl`".

4.1 | Flex Types

The primary purpose of including flex types in the stucture of a cleanpp component is to enable polymorphism without the need for complicated syntax using explicit instances of `std::unique_ptr` or calls to `std::make_unique`. To this end, a flex type provides a special initializer to replace the no-argument initializer of the component's "`_impl`" class that takes one (unused) argument, a variable of the implementation type to be used for this instance of the flex type. Typically,

[3] The exception to this general rule is that when a component is structured with both a kernel and secondary interface, the secondary methods must additionally perform a static cast.

FIGURE 1 Prototypical structure of a Clean++ software component.



this argument is provided at the point of the initializer call by way of a call to the no-argument constructor of the implementing class. Listings 4.1 and 4.2 show some simple client code and a sample flex type.

LISTING 4.1 Client code using `cleanpp::stack`.

```
int main() {
  cleanpp::stack<int> s;
  s.push(4);
  s.push(5);
5 s.push(6);
  int x = s.pop();
  std::cout << s; // prints "<5, 4>"
}
```

LISTING 4.2 Example flex-type for a Clean++ Stack component.

```
namespace cleanpp {

template<typename Item>
class stack: public base {
5 protected:
  template <typename I>
  using _flex_stack_def_t = linked_stack<I>;
  static_assert(std::is_base_of<stack_impl<int>, _flex_stack_def_t<int>>::value,
                "_flex_stack_def_t must derive from stack_impl<Item>");
10

  std::unique_ptr<stack_impl<Item>> rep_;
public:
```

```
    stack() : rep_(std::make_unique<_flex_stack_def_t<Item>>()) {
    }

    template<template<typename> class I>
    stack(__attribute__((unused)) const I<Item>& impl): rep_(std::make_unique<I<Item>>()) {
      static_assert(std::is_base_of<stack_impl<Item>, I<Item>>::value,
              "Template␣parameter␣I␣must␣derive␣from␣stack_impl<Item>");
    }

    stack(const stack<Item> &o) = delete;
    stack(stack<Item>&& o): rep_(std::move(o.rep_)) {
      o.rep_ = std::make_unique<_flex_stack_def_t<Item>>();
    }
    template<template<typename> class I>
    stack(stack<Item>&& o,
          __attribute__((unused)) const I<Item>& impl): rep_(std::move(o.rep_)) {
      static_assert(std::is_base_of<stack_impl<Item>, I<Item>>::value,
                   "Template␣parameter␣I␣must␣derive␣from␣stack_impl<Item>");
      o.rep_ = std::make_unique<I<Item>>();
    }

    stack<Item>& operator=(const stack<Item>& o) = delete;
    stack<Item>& operator=(stack<Item>&& other) {
    if (&other == this) {
      return *this;
    }
    rep_ = std::move(other.rep_);
    other.rep_ = std::make_unique<_flex_stack_def_t<Item>>();
    return *this;
    }
    template<template<typename> class I>
    stack<Item>& operator=(stack<Item>&& other,
                           __attribute__((unused)) const I<Item>& impl) {
      static_assert(std::is_base_of<stack_impl<Item>, I<Item>>::value,
                  "Template␣parameter␣I␣must␣derive␣from␣stack_impl<Item>");
      if (&other == this) {
        return *this;
      }
      rep_ = std::move(other.rep_);
      other.rep_ = std::make_unique<I<Item>>();
      return *this;
    }

    void clear() {
    this->rep_->clear();
    }

```

```
    virtual void push(Item&& x) {
      rep_->push(std::forward<Item>(x));
    }

    virtual Item pop() {
      return rep_->pop();
    }

    virtual bool is_empty() const {
      return rep_->is_empty();
    }

    bool operator==(stack<Item>& other) {
      return *this->rep_ == *other.rep_;
    }

    friend std::ostream& operator<<(std::ostream& out, stack<Item>& o) {
      return out << *o.rep_;
    }
  };
}
```

One important feature of the flex type is that the it identifies a default implementation (line 7 in listing 4.2). The primary reason for this inclusion is that when the flex class is instantiated, information about the implementation type is lost. Therefore, when a `stack` (or other flex type) is move-assigned, because Rule 2 prohibits null references, a new (empty) stack must be initialized in its place and since this requires a class name, a default type is provided. It is important to note that the client need not know about this default type—a client can simply declare a variable of type `stack` (line 2 in listing 4.1). A drawback to the flex type pattern is that a variable of a flex type might have its implementation type changed over the course of its lifetime. Of course, this will not affect the correctness of a program but it might have an impact on performance. When performance is a concern, the client should be aware of the performance profiles of the implementation options and select appropriately—there is always the *option* of using a particular implementation.

A flex type can be easily generated—automatically—from Clean++ components that have both "kernel" and "secondary" interfaces, as well as for those that provide just one interface. Implementing a flex counterpart for a secondary interface involves a static cast of the private variable `rep_` and while the method bodies in that case have some complex syntax, the client code is exceptionally clean. Because a flex type such at `mytype` can be automatically generated from the `mytype_impl` class, a human need not ever look at the "ugly" code in the flex type class.

5 | ILLUSTRATIVE SIMPLE Clean++ COMPONENTS DESIGN AND IMPLEMENTATION

Designing abstract data types (or software components) to support sound local reasoning requires some changes to the usual way of programming in C++. To illustrate the ideas, this section considers a detailed `Stack` example. We begin by noting the inherent aliasing difficulties in the interface for `Stack` from the standard template libraries. Two specific design decisions that complicate modular reasoning are the `push()` method, which copies a reference and creates an

alias, and the `top()` method, which returns an alias to the top of the stack. Minimizing aliases and simplifying reasoning requires a (minor) interface redesign as discussed in this section.

5.1 | Exampe Clean++ Implementation

As a simple example, we present a class implementation of the type `cleanpp::resource` that can be used as the type parameter to one of the several collectionn-type classes in the Clean++ library. This example serves two purposes. First, it demonstrates how a non-template type might be implemented in the Clean++ discipline, and second, it provides a context that should be familiar to C++ programmers—one of resource ownership and transfer thereof.

LISTING 5.1 Implementation of a Clean++ non-template component called `resource`.

```
namespace cleanpp {
  class resource_impl: public base {
  private:
    int i_;
  public:
    resource_impl(const resource_impl& other) = delete;
    resource_impl(resource_impl&& other) {
      i_ = other.i_;
      other.clear();
    }

    resource_impl& operator=(const resource_impl& other) = delete;
    resource_impl& operator=(resource_impl&& other) {
      if (&other == this) {
        return *this;
      }
      i_ = other.i_;
      other.clear();
      return *this;
    }
    resource_impl() {
      i_ = 0;
    }
    resource_impl(int i) {
      i_ = i;
    }

    void clear() override {
      i_ = 0;
    }

    void mutate(int x) {
      i_ *= x;
    }
  };
```

```
}
```

5.2 | Implementation with Move Semantics and std::unique_ptr

One way a stack can be implemented is using a linked list of raw pointers, and with it come the potential for aliases and reasoning complications. With move semantics and `std::unique_ptr`, it is possible to create an efficient linked implementation of a `stack` while avoiding the reasoning pitfalls that are inherent in raw pointer-based software. Using `std::unique_ptr` everywhere that a traditional linked implementation of a stack would use a raw pointer, the complexity of reasoning required by the traditional implementation can be reduced dramatically while maintaining its performance characteristics.

However, a move-based stack implementation on its own is not enough to eliminate reasoning difficulties in software. The type of item contained within the stack must also provide similar benefits. A sample of such a type, `Resource`, is shown and implemented in Listing 5.1.

Consider a simple singly-linked list implementation. Since it typically requires no aliases, the “next” pointer of each node (and the pointer to the first node) can be replaced by a unique pointer. Specifically, a smart pointer type introduced alongside move semantics in C++ 11, `std::unique_ptr`, can be leveraged as a replacement for pointers in places where a programmer knows *a priori* that there will be no aliasing, and to prevent unwanted aliasing everywhere else. A unique pointer enforces at compile time (by the deletion of the copy constructor and copy assignment operator) that there are no aliases to its contents[4].

Furthermore, since the usual behavior of a stack in C++ involves creating aliases to the contents of the stack, the behavior of a stack in Clean++ will necessarily be different than the behavior of the stack in the C++ standard library. Specifically, a call to `top()` on a `std::stack` returns an *alias* to the object at the top of the stack, and a call to `pop()` simply discards the object formerly at the top of the stack. The aliasing problem can be solved simply: eliminate the `top()` method altogether. But that leaves the question of how to access the object at the top of the stack. The solution to this is also simple: the `pop()` operation removes *and returns* the object at the top of the stack. This behavior change is relatively small, so developers familiar with the STL stack will not be totally out of their element working with a Clean++ stack. (Developers familiar with stacks in other languages will immediately be comfortable with this behavior.)

LISTING 5.2 Code for a linked implementation of the Clean++ stack, making extensive use of `std::unique_ptr`.

```
namespace cleanpp {
  template <typename T>
  class stack_impl: public base {
  private:
    class node: base {
    public:
      T contents;
      std::unique_ptr<node> next;

      node(): contents(), next() {}

      node(T&& new_contents):
```

[4] Given the full power of C++, it is technically possible to subvert the guarantee of `std:unique_ptr` through a convoluted series of pointer manipulations. However, this process is sufficiently exotic that it is beyond the scope of ordinary software development and extremely unlikely to happen by accident.

```
      contents(), next() {
15      std::swap(contents, new_contents);
      }

      node(node const &other) = delete;
      node(node&& other):
20    contents(std::move(other.contents)),
      next(std::move(other.next)) {
        other.clear();
      }

25    node& operator=(const node& other) = delete;
      node& operator=(node&& other) {
        if (&other == this) {
          return *this;
        }
30      contents = std::move(other.contents);
        next = std::move(other.next);
        other.clear();
      }

35    void clear() {
        contents = T();
        next.reset();
      }
    };
40
    std::unique_ptr<node> top_ptr_;
  public:
    stack_impl<T>() { }

45  stack_impl<T>(stack_impl<T> const &other) = delete;
    stack_impl<T>(stack_impl<T>&& other):
    top_ptr_(std::move(other.top_ptr_)) {
      other.clear();
    }
50
    stack_impl<T>& operator=(const stack_impl<T>& other) = delete;
    stack_impl<T>& operator=(stack_impl<T>&& other) {
      if (&other == this) {
        return *this;
55    }

      top_ptr_ = std::move(other.top_ptr_);
      other.clear();
    }
60
```

```
    void clear() override {
        if (!is_empty()) {
            top_ptr_.reset();
        }
    }

    void push(T&& x) override {
        top_ptr_ = std::make_unique<stack_node>(std::forward<T>(x), std::move(top_ptr_));
    }

    T pop() override {
        assert(!is_empty());
        T pop = top_ptr_->contents();
        top_ptr_ = top_ptr_->next();
        return std::move(pop);
    }

    bool isEmpty() const override {
      return top_ptr_ == nullptr;
    }
  };
}
```

Of particular interest here are the Pop and Push operations. In contrast to the Stack component from the C++ standard library, the Pop operation returns the object that was removed from the stack (there is no "Top" operation). This is an important decision that enables the elimination of aliases in the Clean++ Stack component because a client need not first acquire a reference (*i.e.*, an alias) to the item at the top of the stack before removing it. Second, the Push operation takes as an argument an rvalue reference to the item to be placed at the top of the stack. The primary consequence of this decision is that the client no longer owns the object they pass to a call to Push. In the client's program, the argument is surrounded by a call to `std::move` to identify this fact, and after the operation the value of the argument is an initial value for its type.

6 | MORE ADVANCED Clean++ IMPLEMENTATIONS

6.1 | Implementing New Components By Reusing Existing Ones

A savvy developer can leverage the reasoning benefits provided by one component by reusing it to easily implement other software components which then exhibit the same nice reasoning properties.

For an example, we consider a reuse of `Stack` component from the previous section. Doubly-linked lists are used to great effect as the underlying data structure for abstractions such as a list with a cursor, which describes a list in which the client can *insert* a new element at the cursor position, *remove* the element at the cursor position, and *advance* or *retreat* the cursor. Navigating a doubly-linked list in this way is extremely efficient, and the data structure provides a natural way of doing so. Unfortunately, by their nature, doubly-linked lists rely on aliases. Since alias avoidance is a key goal of Clean++, it is desirable to implement this data type without any aliases at all.

We can do so using a pair of Stacks, one of which represents the list contents *preceding* the cursor and the other

representing the *remaining* list contents. Advancing or retreating the cursor, then, is done by popping an element from one stack and pushing it onto the other. Insertion and removal, similarly, are done by pushing and element onto or popping one from a stack. We showed above that the push and pop operations on `stack` are efficient. The entire class takes up about 30 SLOC (see Listing 6.1), and exhibits the desired reasoning characteristics of a Clean++ component.

LISTING 6.1 An implementation of `ListWithCursor` built with a pair of Stacks.

```
namespace cleanpp {
  template <class T>
  class ListWithCursor: public base {
  private:
    stack<T> prec_ { };
    stack<T> rem_ { };
  public:
    void advance() {
      T x{ };
      rem_.pop(x);
      prec_.push(x);
    }

    void retreat() {
      T x{ };
      prec_.pop(x);
      rem_.push(x);
    }

    void insert(T& x) {
      prec_.push(x);
    }

    void remove(T& x) {
      prec_.pop(x);
    }

    bool isAtEnd() {
      return rem_.length() == 0;
    }

    bool isAtFront() {
      return prec_.length() == 0;
    }
  };
}
```

6.2 | Implementing Components with Unavoidable Aliasing

Some work has been done recently with respect to verifying behavioral correctness of programs with aliased references [20, 21, 22]. The products of this research are somewhat exotic and require language-level primitives, so until their results are adopted by C++, an alternative is needed in cases where aliases are unavoidable. Such a case is a linked-list implementation of a Queue, with a pointer to both the head of the list (marking the front of the queue) and the last node in the list (marking the tail of the queue). The tail pointer is an alias to the `next` pointer of the second-to-last node in the list. Eliminating the tail pointer solves the aliasing problem, but incurs unacceptable performance penalties. Such a queue can be implemented within the Clean++ discipline because *no aliases will ever leak to the client*, and thus the soundness of modular reasoning is preserved. Listing 6.2 shows such an implementation.

LISTING 6.2 A linked implementation of a queue in the Clean++ discipline.

```
namespace cleanpp {
  template <typename T>
  class queue: public base {
  private:
    class node: base {
    private:
    public:
      T contents;
      std::shared_ptr<node> next;
      node(): contents(), next() {}

      node(T&& new_contents):
      contents(std::move(new_contents)), next() {}

      node(node const &other) = delete;
      node(node&& other):
      contents(std::move(other.contents)),
      next(std::move(other.next)) {
        other.clear();
      }

      node& operator=(const node& other) = delete;
      node& operator=(node&& other) {
        if (&other == this) {
          return *this;
        }
        contents = std::move(other.contents);
        next = std::move(other.next);
        other.clear();
        return *this;
      }

      void clear() {
        contents = T();
        next.reset();
```

```
    }
  };
  std::shared_ptr<node> top_ptr_;
  std::shared_ptr<node> tail_ptr_;
public:
  queue<T>(): top_ptr_(), tail_ptr_() { }

  queue<T>(queue<T> const &other) = delete;
  queue<T>(queue<T>&& other):
  top_ptr_(std::move(other.top_ptr_)),
  tail_ptr_(std::move(other.tail_ptr_)) {
    other.clear();
  }

  queue<T>& operator=(const queue<T>& other) = delete;
  queue<T>& operator=(queue<T>&& other) {
    if (&other == this) {
      return *this;
    }

    top_ptr_ = std::move(other.top_ptr_);
    tail_ptr_ = std::move(other.tail_ptr_);
    other.clear();
    return *this;
  }

  void clear() {
    top_ptr_.reset();
    tail_ptr_.reset();
  }

  void enqueue(T& x) {
    auto new_tail = std::make_shared<node>(std::move(x));
    if (tail_ptr_ != nullptr) {
      tail_ptr_->next = new_tail;
    } else {
      top_ptr_ = new_tail;
    }
    // Alias!!
    tail_ptr_ = new_tail;
  }

  void dequeue(T& x) {
    std::swap(x, top_ptr_->contents);
    std::swap(top_ptr_, top_ptr_->next);
  }
```

```
    bool isEmpty() const {
      return top_ptr_ == nullptr;
85    }
  };
}
```

The similarities to the linked implementation of `cleanpp::stack` are significant, with the obvious difference that `std::shared_ptr` is used in places where `std::unique_ptr` was used before. This difference is because of the alias introduced at line 75 that is a consequence of maintaining a pointer to the tail of the queue in addition to the head. Importantly, this alias is never leaked to the client (nor is the null reference indicating the end of the queue). A shared pointer is used rather than a raw pointer for several reasons: First, to advertise the fact that this variable might have aliases at any given time, and second to let the C++ compiler manage memory for the programmer.

7 | EVALUATING Clean++

The Clean++ discipline was presented to an undergraduate audience in order to evaluate its efficacy and usability in two kinds of programs: "client" programs that make use of existing Clean++ software components and "implementation" programs that comprise the totality of an implementation of a Clean++ software component. Evaluation showed that the syntactic burden is not overwhelming; it it easy for an undergraduate student to successfully incorporate the move-semantics related into both kinds of programs. However, it quickly became clear that leveraging an existing C++ compiler to produce errors when the discipline is violated was crucial to efficient development, which justified the attention paid to such features from the beginning.

Feedback from evaluation of the discipline informed several descisions in its refinement. To give one example, at several points during evaluation it was deemed necessary to write a function that both changed the value of one of its arguments and returned a separate object. However, because move semantics was a key part of Clean++ from the beginning, this led to the inclusion of Rule 4, by which every method has a return type of `std::tuple`.

8 | RELATED WORK

While the reasoning difficulties aliases pose are well-known, efforts to enhance popular programming languages to keep them under control at the language level have been few and far between. Some efforts have focused on efficient ways to enrich *value types* so that they may be used for as much software as possible or on developing disciplines and style guidelines within popular, existing languages to control and advertise the use of aliases. Other research efforts have been directed toward the development of entirely new languages built from the ground up to eliminate aliasing concerns. Ultimately, not all aliasing can be avoided, so admitting aliasing and enabling sound reasoning in its presence are topics that have received attention in the formal methods literature.

8.1 | The Rust Programming Language

Rust, an open-source programming language project sponsored by Mozilla, is a relatively new and popular programming language that aims to (and does, in many cases) solve exactly the kind of problems identified in this paper and addressed by Clean++. However, the motivations for it are related to memory safety, not based in a desire for simple (abstract) reasoning.

Specifically, Rust identifies that shared data (e.g., in the form of aliases) is a big problem for compilers. It aims to solve this problem by introducing a statically-checked system of ownership and borrowing to ensure that immutable data is never mutated except when the programmer really, really wants that behavior. Rust's system of ownership and borrowing have deep implications for alias control in programs and is quite complicated, so it will be discussed only at a high level here. Ownership in Rust follows a few simple rules:

1. Each value in Rust has a variable called its owner
2. A value may have only one such owner at a time
3. When an owner goes out of scope, its value is dropped (i.e., its memory is freed)

Assignment in Rust, by default, *transfers ownership* (i.e., *moves*) from the right hand side to the left hand side. The semantics of Rust's ownership transfer is similar—though not identical—to the semantics of assignment with the `std::move` operation in C++ (and is quite different than the intended meaning of move in Clean++). The difference is that in Rust, a compile-time check ensures that no moved variable is subsequently used and C++ makes no such check. In Clean++, as discussed above, the approach is opposite Rust: values that have been moved from *can* be used, and their value is well-defined.

Rust takes a different approach than Clean++ to situations in which shared data is unavoidable. In Rust, a variable may *borrow* ownership from another, in which case the second variable is a *reference* to the first—it does not introduce an alias on its own (see Figure 2). Borrows may be done mutably or immutably, and the rules for each vary. Overall, borrowing in Rust follows three rules:

1. There may be either one mutable reference or any number of immutable references in scope
2. A reference must always be valid (the value it refers to must not have been dropped)
3. Assignment may only be done to variables that are not borrowed from

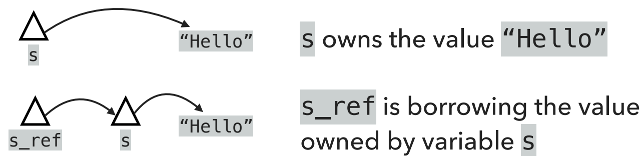


FIGURE 2 Borrowing in Rust

In Clean++, on the other hand, aliasing is advertised through the use of `std::shared_ptr`. This enables a Clean++ programmer to quickly recognize those places where aliasing occurs and to be careful when reasoning about them. Rust's approach is similar in this regard: shared data is advertised (through the borrowing operator `&`) when needed so that the programmer can reason more carefully. However, by virtue of being an entirely new programming language, Rust is able to offer additional compile-time guarantees beyond what Clean++ is capable of. Specifically, the ownership and borrowing rules are enforced at *compile time*. That is not to say that Clean++ is without any copmile-time checks. Several of the idioms introduced above result in code that is flagged as erroneous by a standard C++ compiler, and many of the other rules could be statically analyzed by a custom analyzer for Clean++.

8.1.1 | Clean++ vs. Rust

Because Rust addresses many of the problems identified and addressed by Clean++, it is natural to ask whether Clean++ has already been made redundant by Rust. The answer is no. First, while Rust is an entirely new language that requires programmers to learn new syntax, introduces complex new ideas[5], and will need to be validiated in large software projects, Clean++ is simply a discipline of programming in a well-established programming language that virtually all programmers are familiar with.

Additionally, while Rust provides limited facilities for true object-oriented programming, C++ provides a full suite of capabilities for building robust, modular, object-oriented programs that make use of abstract data types, polymorphism, and inheritance. These capabilities are drawn upon in Clean++ to maintain client-implementer separation and to encourage the use of many levels of abstraction—things that would, in Rust, be at best difficult and cumbersome to achieve. For comparison's sake, the appendix contains a sample component written both in Clean++ and Rust.

Finally, Rust's solutions to many of the problems addressed in this paper are framed in terms of memory management and memory safety—*not* ease of reasoning. This does not mean that Rust's fixes do not improve reasoning (they do), but it does mean that in some cases Rust may not go "far enough" toward easy reasoning to make it amenable to automated formal verification. We believe that Clean++ does, since it is inspired directly by verification-focused languages such as RESOLVE [17]. By including comment-based formal contracts, it is believed that programs written from the ground up in Clean++ could be automatically verified to be correct.

8.2 | Google's Stylistic Suggestions for Writing Good Code in C++

Google's style guide [23] offers extensive guidance on how to write good C++ code, from variable naming conventions to high-level structuring. Several sections address themes directly related to this paper.

Of most relevance to the topic is the advice on "Ownership and Smart Pointers". In this section, an argument is made as to why having a regimented discipline for expressing ownership (beyond simple pointers) is a Good Thing in C++ programs. The stylistic suggestion made in the guide is to "prefer to keep ownership with the code that allocated it," and when ownership transfer is necessary, to use `std::unique_ptr` to make such transfer explicit.

Two other sections, "Copyable and Movable Types" and "Rvalue References" are also closely related to the discipline proposed in this paper. Google's C++ style guide recommends that a class be either copyable or movable or neither, but not both copyable and movable except in certain situations. The justification given for this rule is that having a copyable class that is also movable is a potential source of bugs because the expected (i.e., copying) behavior of parameter values or return values might not actually occur. The guide argues that defining moving operations on a copyable class should be viewed strictly as a performance optimization and so should not be used unless there is a clear performance improvement for doing so. The section on rvalue references makes the recommendation that rvalue references be used in overloaded function pairs (one taking a `const&` and the other a `&&` argument). Again, the crux of the decision is that rvalue references and move semantics should be viewed strictly as a performance optimization. Minimizing aliasing and simplifying reasoning are not considered in the discussion.

[5] The ownership system alone has been the topic of discussion for several weekly meetings of a group of university professors and PhD students...and it still is not completely understood by all of them.

8.3 | Rich, Efficient Value Types and Apple's Swift

One typical solution to the aliasing problem offered by modern programming languages is the introduction of rich and efficient *value types* that, when assigned, copy the variable's *entire value* to the destination.[6] The obvious advantage of value types in the context of this paper is that aliases are never introduced, thus enabling simpler reasoning. Unfortunately, frequent copying has the potential to be quite costly for large types. This has led some languages to impose rules on what may be done with variables of value types.

For example, Apple's newest programming language, Swift [24], offers a rich toolbox for creating value types (called `structs`). However, value types in Swift are immutable by default and the compiler attempts to restrict value types from being mutated without additional annotations. The reason for this limitation is that structs in Swift are copied lazily (sometimes called "copy-on-write"), so making them immutable allows Swift to offer equivalent performance in value-based code as it does in reference-based code (except in rare cases where a struct is mutated). Implementing lazy copying at the language level ensures that structs are used in contexts where they are not mutated and encourages programmers to consider using value types where appropriate.

8.4 | Functional Languages and Immutable Data

A key insight of functional languages is that the exclusive use of immutable data can eliminate a whole class of reasoning problems. Immutable types, of course, are not limited to functional languages; as mentioned above, value types in Swift are immutable by default. Functional languages' *exclusive* use of immutable types is what makes them so attractive as reasoning-focused languages[25]. While functional programming has its benefits, the focus of this paper is on developing practical principles for C++.

8.5 | Pointer-Free Programming in ParaSail

As mentioned in the introduction, recent work by Tucker Taft on ParaSail [13] addresses several of the issues raised in this paper, but in the context of a new language. ParaSail was developed with two complementary goals in mind: pointer-free and parallel programming. These two goals are complementary because reasoning in the presence of parallelism is made significantly more complicated when aliases are introduced—indeed, parallel programs often rely on data sharing between threads in the first place, so introducing unnecessary aliases makes things much worse. ParaSail unifies pointer-free and parallel programming idioms by permitting the sharing of data across threads explicitly only in cases where it is intended. Everywhere else there are no aliases.

8.6 | Language Design to Prevent Aliases

Beyond ParaSail, there have been attempts to design programming languages keeping in mind the principles discussed in this paper. In particular, RESOLVE [17, 26] has been used in large-scale efforts to automatically verify the full functional correctness of imperative programs. RESOLVE, an integrated specification-programming language, was designed with *clean semantics*; that is, there are no references in RESOLVE and, thus, no aliases. Every variable is reasoned about using its *abstract value*. Furthermore, by employing a *swapping paradigm* at the language level [10], a variable in a RESOLVE program is guaranteed to be the sole owner of its data at any given time, and copying is rare. These two factors combine to make it relatively simple to reason about programs written in RESOLVE because many

[6] At least, this is the client's view of what is happening. Some types, such as immutable ones, need not be copied entirely.

of the low-level memory management details can be abstracted away. This is perhaps the central contribution of the RESOLVE project: a recognition that if every pointer points to a distinct thing, then there is no need to reason about pointers at all.

RESOLVE design principles have been proposed for developing software in C++ and Java. These prior solutions aim to "simulate" RESOLVE-style programming in these popular languages, and do not leverage move semantics or other C++ 11 features. Zaccai, in his PhD thesis, discusses the challenges to automated verification in Java which stem, primarily, from aliased references (and arguments) [9]. Hollingsworth, *et al*, build relatively large-scale software using a dialect of C++ called RESOLVE/C++ [16]. Their work has validated that certain formal methods principles—such as the ones presented here—can be applied at scale in real software.

8.7 | Object Ownership Systems

One popular approach to dealing with aliases in a software verification context is object ownership systems [27, 28, 29, 2]. Such systems impose a structure and restrictions the heap and force programs to behave a certain way; in particular, behavior in which aliases are managed properly. Ownership systems are cast as extensions of typing and are therefore checkable by relatively simple static analyzers.

8.8 | Specification and Verification with Aliasing

This paper has presented a discipline in which aliases are largely avoided at all costs, primarily through extensive use of `std::unique_ptr`. However, as discussed briefly above, data sharing—including aliasing—is occasionally unavoidable. The challenge in software verification (i.e., reasoning) involving objects, aliasing, and properties about the heap is well recognized in the literature [7, 8].

A variety of efforts have addressed the verification challenge, but if aliasing is minimized, then reasoning will be simplified when using any of the approaches, including ones based on separation logic [3, 30, 31, 32], dynamic frames[33, 34, 35, 36, 37, 38], or region logic[39]. Verification efforts based on devising and reusing concepts in which a set of locations (an abstraction of addresses) is "shared" [20, 21, 22] will also be simplified. The discussion in [20] also contains a solution to avoid the classical parameter aliasing problem on calls with repeated arguments, a solution that can be realized in C++ with move semantics.

9 | FUTURE WORK

Clean++ is a developing entity, so there are a number of avenues for work in Clean++. For example, at present, Clean++ does not have provisions for concurrent programming. Validating automatically that Clean++ code adheres to the discipline remains an important goal. Finally, there are avenues for exploration in how Clean++ can be used to simplify and teach formal reasoning in a potentially more familiar environment for students than current reasoning-focused languages and tools.

9.1 | Concurrency

The Clean++ discipline does not immediately support using components in concurrent contexts. Specifically, the claim was made above that "passing by reference does, of course, introduce an alias (since the calling program and

the method body both hold references to the same object), but it not a *dangerous* one because the aliased references are never in scope at the same time." This claim is clearly not true for parallel programs, but in such cases the sharing of data is not only unavoidable but actually desired. Discussion of extending Clean++ to support parallel programs is a topic for further research; such development is a natural consequence of current research of the authors [40, 41] and others [13].

9.2 | Automated Clean++ Validation

The Clean++ discipline, as presented here, is rigid enough in many respects that a static analyzer could be developed to check automatically that code adheres to several of the core ideas.

There are several aspects of the discipline that would be relatively easy to check and others that could prove more challenging. Specifically, it would be trivial to check for any raw pointers and warn the programmer if any are found. If a smart pointer other than `std::unique_ptr` or `std::shared_ptr` is found, a warning would also be dispatched suggesting that the programmer employ one of these two "acceptable" types. Ensuring that all user-defined types delete the copy constructor and assignment operator and have replaced them with a move constructor and move assignment operator would not be difficult. Finally, any instances of null reference "leakage" to the client could be caught with only a moderatlely sophisticated static analysis.

Challenges to automating Clean++ validation include ensuring that the no-argument initializer actually creates an object with a valid initial value. Options for enabling this kind of sophisticated analysis include employing comment-based specifications regarding abstraction and initial values. In most cases, default initializers are straightforward and the computation of the abstract value generated by such a method, given an appropriate specification, could be achieved.

9.3 | Education

The benefits afforded by Clean++ code make it a good candidate to be a tool for teaching formal reasoning to CS students. In Clean++, a programmer explicitly advertises the moving behavior of components and forces the client to use rvalue references for arguments to component methods. Thus, she never needs to reason about the possibility of aliases—every object always has a unique owner. This is a huge benefit, especially to students who are only beginning to learn about programming.

Another benefit of the strictness of the Clean++ discipline with regard to the ease of reasoning is that many pieces of code that introduce aliases are flagged as erroneous by any C++ compiler, as long as the programmer is using Clean++ components. This feature alone could help beginning students see how to write programs that avoid aliases, and to develop an understanding of the mechanisms of data transfer that are used in many of today's popular programming languages.

10 | CONCLUSIONS

Despite the common view of move semantics as a purely performance-improving trick, there is significant potential for move semantics to be broadly applied in large software projects not only to improve performance but also to simplify reasoning and, thus, increase the robustness of software systems. The discipline presented here, Clean++, is only part of what is required to reap all of the benefits of clean semantics in C++ software. Clean semantics and its associated

reasoning benefits can be realized through judicious—and careful—use of C++ move semantics and `std::unique_ptr`. Future efforts involve tool development to assist and simplify such software development.

Repository of Clean++ Components

A small proof-of-concept library of software components has been written in the Clean++ discipline to demonstrate its utility, flexibility, and efficiency. The library is available on the authors' GitHub [42].

Acknowledgements

The authors thank all of the members of RSRG at The Ohio State University, Clemson University, and other institutions for their valuable feedback about this work. They especially thank their undergraduate evaluator, Will Janning, and appreciate his efforts and time spent learning the Clean++ discipline and writing hundreds of lines of code.

references

[1] Filipović I, O'Hearn P, Torp-Smith N, Yang H. Blaming the client: on data refinement in the presence of pointers. Formal Aspects of Computing 2010;22(5):547–583. http://dx.doi.org/10.1007/s00165-009-0125-8.

[2] Boyapati C, Liskov B, Shrira L. Ownership Types for Object Encapsulation. In: Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages POPL '03, New York, NY, USA: ACM; 2003. p. 213–223. http://doi.acm.org/10.1145/604131.604156.

[3] O'Hearn PW. Resources, Concurrency, and Local Reasoning. Theor Comput Sci 2007 Apr;375(1-3):271–307. http://dx.doi.org/10.1016/j.tcs.2006.12.035.

[4] Hogg J, Lea D, Wills A, deChampeaux D, Holt R. The Geneva Convention on the Treatment of Object Aliasing. SIGPLAN OOPS Mess 1992 Apr;3(2):11–16. http://doi.acm.org/10.1145/130943.130947.

[5] Müller P, Poetzsch-Heffter A. Foundations of Component-based Systems. New York, NY, USA: Cambridge University Press; 2000.p. 137–159. http://dl.acm.org/citation.cfm?id=336431.336449.

[6] Weide BW, Heym WD. Specification and Verification with References. In: Proceedings OOPSLA Workshop on Specification and Verification of Component-Based Systems; 2001. p. 50–59.

[7] Leavens GT, Leino KRM, Müller P. Specification and Verification Challenges for Sequential Object-oriented Programs. Form Asp Comput 2007 Jun;19(2):159–189. http://dx.doi.org/10.1007/s00165-007-0026-7.

[8] Reynolds JC. Separation Logic: A Logic for Shared Mutable Data Structures. In: Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science LICS '02, Washington, DC, USA: IEEE Computer Society; 2002. p. 55–74. http://dl.acm.org/citation.cfm?id=645683.664578.

[9] Zaccai DS. A Balanced Verification Effort for the Java Language. PhD thesis, Ohio State University; 2016.

[10] Harms DE, Weide BW. Copying and Swapping: Influences on the Design of Reusable Software Components. IEEE Trans Softw Eng 1991 May;17(5):424–435. http://dx.doi.org/10.1109/32.90445.

[11] Kulczycki G, Sitaraman M, Krone J, Hollingsworth JE, Ogden WF, Weide BW, et al. A Language for Building Verified Software Components. In: Safe and Secure Software Reuse - 13th International Conference on Software Reuse, ICSR 2013, Pisa, Italy, June 18-20. Proceedings; 2013. p. 308–314. https://doi.org/10.1007/978-3-642-38977-1_23.

[12] Castegren E, Wrigstad T, Kappa: Insights, Current Status and Future Work; 2016. International Workshop on Aliasing, Confinement and Ownership (IWACO).

[13] Taft ST. ParaSail: A Pointer-Free Pervasively-Parallel Language for Irregular Computations. The Art, Science, and Engineering of Programming 2019;Volume 3. http://arxiv.org/abs/1902.00525.

[14] Hinnant HE, Dimov P, Abrahams D, A Proposal to Add Move Semantics Support to the C++ Language; 2002. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2002/n1377.htm.

[15] Klabnik S, Nichols C. The Rust Programming Language. San Francisco, CA, USA: No Starch Press; 2018.

[16] Hollingsworth JE, Blankenship L, Weide BW. Experience Report: Using RESOLVE/C++ for Commercial Software. In: Proceedings of the 8th ACM SIGSOFT International Symposium on Foundations of Software Engineering: Twenty-first Century Applications SIGSOFT '00/FSE-8, New York, NY, USA: ACM; 2000. p. 11–19. http://doi.acm.org/10.1145/355045.355048.

[17] Sitariman M, Weide B. Component-based Software Using RESOLVE. SIGSOFT Softw Eng Notes 1994 Oct;19(4):21–22. http://doi.acm.org/10.1145/190679.199221.

[18] Cheon Y, Leavens G, Sitaraman M, Edwards S. Model variables: cleanly supporting abstraction in design by contract. Software: Practice and Experience 2005;35(6):583–599. https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.649.

[19] Weide BW, Edwards SH, Heym WD, Long TJ, Ogden WF. Characterizing observability and controllability of software components. In: Proceedings of Fourth IEEE International Conference on Software Reuse; 1996. p. 62–71.

[20] Kulczycki G. Direct Reasoning. Phd dissertation, Clemson University; 2004.

[21] Kulczycki G, Sitaraman M, Weide BW, Rountev A. A Specification-based Approach to Reasoning About Pointers. In: Proceedings of the 2005 Conference on Specification and Verification of Component-based Systems SAVCBS '05, New York, NY, USA: ACM; 2005. http://doi.acm.org/10.1145/1123058.1123066.

[22] Sun YS, Weide BW, Zaccai D, Sivilotti PAG, Sitaraman M. The RESOLVE Approach for Achieving Modular Verification: Progress and Challenges in Addressing Aliasing. Clemson, SC 29634: Clemson University - School of Computing; 2017.

[23] Google C++ Style Guide; 2018. https://google.github.io/styleguide/cppguide.html.

[24] Apple Inc . The Swift Programming Language. Swift Programming Series, Apple Inc.; 2018.

[25] Stump A. Verified Functional Programming in Agda. New York, NY, USA: Association for Computing Machinery and Morgan & Claypool; 2016.

[26] Sitaraman M, Adcock B, Avigad J, Bronish D, Bucci P, Frazier D, et al. Building a push-button RESOLVE verifier: Progress and challenges. Formal Aspects of Computing 2011;23(5):607–626. http://dx.doi.org/10.1007/s00165-010-0154-3.

[27] Dietl W, Müller P. Aliasing in Object-Oriented Programming. Berlin, Heidelberg: Springer-Verlag; 2013.p. 289–318. http://dl.acm.org/citation.cfm?id=2554511.2554528.

[28] Cameron NR, Drossopoulou S, Noble J, Smith MJ. Multiple Ownership. In: Proceedings of the 22Nd Annual ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications OOPSLA '07, New York, NY, USA: ACM; 2007. p. 441–460. http://doi.acm.org/10.1145/1297027.1297060.

[29] Cameron N, Drossopoulou S, Noble J. Aliasing in Object-Oriented Programming. Berlin, Heidelberg: Springer-Verlag; 2013.p. 84–108. http://dl.acm.org/citation.cfm?id=2554511.2554518.

[30] O'Hearn PW, Reynolds JC, Yang H. Local Reasoning About Programs That Alter Data Structures. In: Proceedings of the 15th International Workshop on Computer Science Logic CSL '01, London, UK: Springer-Verlag; 2001. p. 1–19. http://dl.acm.org/citation.cfm?id=647851.737404.

[31] O'Hearn PW, Yang H, Reynolds JC. Separation and Information Hiding. In: Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages POPL '04, New York, NY, USA: ACM; 2004. p. 268–280. http://doi.acm.org/10.1145/964001.964024.

[32] Piskac R, Wies T, Zufferey D. Automating Separation Logic with Trees and Data. In: Proceedings of the 16th International Conference on Computer Aided Verification - Volume 8559 New York, NY, USA: Springer-Verlag New York, Inc.; 2014. p. 711–728. http://dx.doi.org/10.1007/978-3-319-08867-9_47.

[33] Bruns D, Mostowski W, Ulbrich M. Implementation-level verification of algorithms with KeY. International Journal on Software Tools for Technology Transfer 2015;17(6):729–744. http://dx.doi.org/10.1007/s10009-013-0293-y.

[34] Kassios IT. Dynamic Frames: Support for Framing, Dependencies and Sharing Without Restrictions. In: Proceedings of the 14th International Conference on Formal Methods FM'06, Berlin, Heidelberg: Springer-Verlag; 2006. p. 268–283. http://dx.doi.org/10.1007/11813040_19.

[35] Kassios IT, Dynamic Frames and Automated Verification; 2011.

[36] Kassios IT. The Dynamic Frames Theory. Form Asp Comput 2011 May;23(3):267–288. http://dx.doi.org/10.1007/s00165-010-0152-5.

[37] Leino KRM. Specification and verification of object-oriented software. In: Broy M, Sitou W, Hoare T, editors. Engineering Methods and Tools for Software Safety and Security Volume 22 NATO Science for Peace and Security Series - D: Information and Communication Security, IOS Press; 2009. p. 231–266.

[38] Schmitt PH, Ulbrich M, WeißB. Dynamic Frames in Java Dynamic Logic. In: Proceedings of the 2010 International Conference on Formal Verification of Object-oriented Software FoVeOOS'10, Berlin, Heidelberg: Springer-Verlag; 2011. p. 138–152. http://dl.acm.org/citation.cfm?id=1949303.1949313.

[39] Banerjee A, Naumann DA, Rosenberg S. Regional Logic for Local Reasoning About Global Invariants. In: Proceedings of the 22Nd European Conference on Object-Oriented Programming ECOOP '08, Berlin, Heidelberg: Springer-Verlag; 2008. p. 387–411. http://dx.doi.org/10.1007/978-3-540-70592-5_17.

[40] Weide A. Enabling Modular Verification of Concurrent Programs; 2017, european Conference on Object-Oriented Programming, Doctoral Symposium.

[41] Weide A, Sivilotti P, Sitaraman M. An Array Abstraction to Amortize Reasoning About Parallel Client Code. In: Computing Conference London, UK: Springer; 2021. .

[42] Weide A, Sivilotti P, Sitaraman M, Clean++ Library; 2021. https://github.com/osu-rsrg/cleanpplib.

A | OBJECT-ORIENTED PROGRAMMING IN RUST VS. Clean++

Consider a simple type, called `Natural` that is modeled by the natural numbers $\mathbb{N}$ and is unbounded (this is similar to a “Big Integer” found in libraries for many languages). Our implementation, however, will be represented by a single `unsigned long` (or `u64` in Rust) for the sake of simplicity, so it is bounded in reality. Keep in mind while reading this that an alternative implementation (e.g., based on a `Stack`) could be employed to make it unbounded.

LISTING A.1 Relatively small program approximating the kind of modular component design preferred by Clean++ in Rust.

```
pub mod natural {

    const RADIX: i64 = 10;
    pub trait NaturalNumberKernel {
        fn multiply_by_radix(&mut self, digit: i64);
        fn divide_by_radix(&mut self, digit: &mut i64);
        fn is_zero(&self) -> bool;
        fn clear(&mut self);
    }

    pub trait NaturalNumber: NaturalNumberKernel {
        fn increment(&mut self) {
            let mut last_digit = 0;
            self.divide_by_radix(&mut last_digit);
            last_digit += 1;
            if last_digit == RADIX {
                last_digit -= RADIX;
                self.increment();
            }
            self.multiply_by_radix(last_digit);
        }

        fn decrement(&mut self) {
            let mut last_digit = 0;
            self.divide_by_radix(&mut last_digit);
            if last_digit == 0 {
                last_digit += RADIX;
                self.decrement();
            }
            last_digit -= 1;
            self.multiply_by_radix(last_digit);
        }

        fn set_from_i64(&mut self, mut n: i64) {
            self.clear();
            if n > 0 {
                let d = n % RADIX;
                n /= RADIX;
```

```
                self.set_from_i64(n);
40              self.multiply_by_radix(d);
            }
        }
    }

45  pub fn add(lhs: &mut Box<dyn NaturalNumber>, rhs: &mut Box<dyn NaturalNumber>) {
        let mut lhs_last = 0;
        let mut rhs_last = 0;
        lhs.divide_by_radix(&mut lhs_last);
        rhs.divide_by_radix(&mut rhs_last);
50      lhs_last += rhs_last;
        if lhs_last > RADIX {
            lhs_last -= RADIX;
            lhs.increment();
        }
55      if !rhs.is_zero() {
            add(lhs, rhs);
        }
        lhs.multiply_by_radix(lhs_last);
        rhs.multiply_by_radix(rhs_last);
60  }

    pub fn subtract(lhs: &mut Box<dyn NaturalNumber>, rhs: &mut Box<dyn NaturalNumber>) {
        let mut lhs_last = 0;
        let mut rhs_last = 0;
65      lhs.divide_by_radix(&mut lhs_last);
        rhs.divide_by_radix(&mut rhs_last);
        lhs_last -= rhs_last;
        if lhs_last < 0 {
            lhs_last += RADIX;
70          lhs.decrement();
        }
        if !rhs.is_zero() {
            subtract(lhs, rhs);
        }
75      lhs.multiply_by_radix(lhs_last);
        rhs.multiply_by_radix(rhs_last);
    }

    pub struct Bounded {
80      n: i64,
    }

    impl Bounded {
        pub fn new() -> Bounded {
85          Bounded { n: 0 }
```

```
        }
    }

    impl NaturalNumberKernel for Bounded {
        fn multiply_by_radix(&mut self, digit: i64) {
            self.n *= RADIX;
            self.n += digit;
        }
        fn divide_by_radix(&mut self, digit: &mut i64) {
            *digit = self.n % RADIX;
            self.n /= RADIX;
        }
        fn is_zero(&self) -> bool {
            self.n == 0
        }
        fn clear(&mut self) {
            self.n = 0;
        }
    }

    impl NaturalNumber for Bounded {
        fn increment(&mut self) {
            self.n += 1;
        }

        fn decrement(&mut self) {
            self.n -= 1;
        }
        fn set_from_i64(&mut self, n: i64) {
            self.n = n;
        }
    }
}

fn main() {
    use natural::{add, Bounded, NaturalNumber};
    let mut n: Box<dyn NaturalNumber> = Box::new(Bounded::new());
    n.set_from_i64(42);
    let mut m: Box<dyn NaturalNumber> = Box::new(Bounded::new());
    m.set_from_i64(21);
    add(&mut n, m);
    // n = 63, m = 21
}
```

LISTING A.2 The Clean++ version of the Rust "natural" component above. (`#include` statements are elided.)

```
/*
 * --------------------------------
```

```
 * File: natural_number_kernel.hpp
 * --------------------------------
 */

namespace cleanpp::natural {
class natural_number_kernel: public clean_base {
public:
    static const int RADIX = 10;

    virtual bool is_zero() = 0;
    virtual void multiply_by_radix(int digit) = 0;
    virtual void divide_by_radix(int &digit) = 0;
};
}

/*
 * -------------------------
 * File: natural_number.hpp
 * -------------------------
 */

namespace cleanpp::natural {
class natural_number: public natural_number_kernel {
public:
    virtual void increment();
    virtual void decrement();
    virtual void set_from_int(int n);
};

void add(std::unique_ptr<natural_number> &x, std::unique_ptr<natural_number> &y);
void subtract(std::unique_ptr<natural_number> &x, std::unique_ptr<natural_number> &y);
}


/*
 * -------------------------
 * File: natural_number.cpp
 * -------------------------
 */

namespace cleanpp::natural {
void natural_number::increment() {
    int d = 0;
    this->divide_by_radix(d);
    d++;
    if (d == RADIX) {
        d -= RADIX;
        this->increment();
```

```
50      }
        this->multiply_by_radix(d);
    }

    void natural_number::decrement() {
55      assert(!is_zero());
        int d = 0;
        this->divide_by_radix(d);
        d--;
        if (d < 0) {
60          d += RADIX;
            this->decrement();
        }
        this->multiply_by_radix(d);
    }
65
    void natural_number::set_from_int(int n) {
        assert(n >= 0);
        if (n == 0) {
            this->clear();
70      } else {
            int nLeft = n / RADIX;
            this->set_from_int(nLeft);
            this->multiply_by_radix(n % RADIX);
        }
75  }

    void add(std::unique_ptr<natural_number> &x, std::unique_ptr<natural_number> &y) {
        int x_low;
        x->divide_by_radix(x_low);
80      int y_low;
        y->divide_by_radix(y_low);
        if (!y->is_zero()) {
            add(x, y);
        }
85      x_low += y_low;
        if (x_low >= natural_number::RADIX) {
            x_low -= natural_number::RADIX;
            x->increment();
        }
90      x->multiply_by_radix(x_low);
        y->multiply_by_radix(y_low);
    }

    void subtract(std::unique_ptr<natural_number> &x, std::unique_ptr<natural_number> &y) {
95      int x_low;
        x->divide_by_radix(x_low);
```

```
        int y_low;
        y->divide_by_radix(y_low);
        if (!y->is_zero()) {
100         subtract(x, y);
        }
        x_low -= y_low;
        if (x_low < 0) {
            x_low += natural_number::RADIX;
105         x->decrement();
        }
        x->multiply_by_radix(x_low);
        y->multiply_by_radix(y_low);
    }
110 }

    /*
     * ----------------------
     * File: bounded_nn.cpp
115  * ----------------------
     */

    namespace cleanpp::natural {
    bounded_nn::bounded_nn(int n): n(n) {};
120
    bounded_nn::bounded_nn(bounded_nn&& other): n(std::move(other.n)) {
        other.clear();
    }

125 bounded_nn& bounded_nn::operator=(bounded_nn&& other) {
        if (&other == this) {
            return *this;
        }

130     this->n = other.n;
        other.clear();
        return *this;
    }

135 bool bounded_nn::operator==(const bounded_nn &other) {
        return this->n == other.n;
    }

    void bounded_nn::clear() {
140     this->n = 0;
    }
    bool bounded_nn::is_zero() {
        return this->n == 0;
```

```
}
void bounded_nn::multiply_by_radix(int d) {
    this->n *= RADIX;
    this->n += d;
}
void bounded_nn::divide_by_radix(int &d) {
    d = this->n % RADIX;
    this->n /= RADIX;
}

void bounded_nn::increment() {
    this->n++;
}

void bounded_nn::decrement() {
    this->n--;
}

void bounded_nn::set_from_int(int n) {
    this->n = n;
}
}

/*
 * -------------
 * File: main.cpp
 * -------------
 */

using namespace cleanpp::natural;

int main() {
    std::unique_ptr<natural_number> n = std::make_unique<bounded_nn>();
    n->set_from_int(42);
    std::unique_ptr<natural_number> m = std::make_unique<bounded_nn>();
    m->set_from_int(21);
    add(n, m);
    // n = 63, m = 21
}
```

The Rust version of this program mirrors the structure of its counterpart in Clean++, however there are some important differences. First, whereas `std::unique_ptr<T>` is used to ensure unique ownership in Clean++, Rust provides the `Box<T>` type to denote unique ownership of dynamic data structures. The two types are similar in more ways than not, and for our purposes they are interchangeable when it comes to reasoning except for the fact that Rust provides some memory safety guarantees that are not present in Clean++ with `std::unique_ptr<T>`. Second, while C++ permits a class to have a variety of intializers with varying parameter lists, Rust does not permit such function overloading (in fact, initializers in Rust are not special; they are simply regular functions with the conventional name “new”).