AdaCore TECH PAPER

MISSION CRITICAL RUST: Managing Memory

Mission Critical Rust: Managing Memory

Executive Summary

Pointer errors have plagued software developers for decades. Rust's innovative and expressive approach helps make pointers safe and efficient.

Pointers have been a staple of programming languages since the earliest days of computing, serving two purposes:

- **Indirection:** a means to share (rather than copy) data values within a program. This can be implicit, for example by passing a variable as a "by reference" parameter, or explicit through syntax for reference creating / dereferencing (such as "&x" and "*p" in C).
- **Dynamic allocation:** a means to construct and manipulate data structures (linked lists, trees, graphs, ...) that can grow and shrink during program execution.

However, with power comes danger, and pointers can compromise both safety and performance. This paper investigates the challenges that pointers bring and explains how Rust meets these challenges. Rust's approach requires programmers to think about pointers in a new way, but the effect is memory safety / early detection of pointer errors and efficient performance, with the expressiveness of a general-purpose data structuring facility.

Indirection issues

Allowing a pointer to outlive the value it points to creates an insidious bug known as a dangling reference. For example, if a C function returns a reference to one of its local variables and then accesses the value referenced by the function's result, the effect is undefined. The error can be hard to diagnose, since any anomalous behavior is manifest not when the dangling reference is created but instead at some later point when the invalid value is used.

Unprotected access to a value shared across multiple threads (a "data race") is equally dangerous. Without some protection ensuring mutual exclusion, the value can be corrupted if it is being assigned by one thread while being either assigned or read by another thread. As with dangling references, the effect is not necessarily localized to the construct where the error occurs. Debugging concurrent code is challenging in general because of the multiple possible control paths; data races raise the ante considerably.

Performance can also take a hit since indirect accesses complicate the analysis needed for optimizations. Aliasing in the following C program illustrates the problem:

```
#include <stdio.h>
int func(const int *p, int *q){
   (*q)++;
   return *p + *p;
}
void main(){
   int n = 100;
   printf("func() returns %d\n", func(&n, &n)); // 202
}
```

An optimizing compiler would like to store `*p` in a register on entry to `func()`, for use in the return statement. However, this would require the (invalid) assumption that `*p` and `*q` do not overlap.

Dynamic allocation issues

Many applications require data structures whose size can grow and shrink dynamically, with no a *priori* limit on the maximum size. Programming languages accommodate this requirement through dynamic allocation in a storage area known as the heap, but that raises the issue of when and how to reclaim storage that is no longer accessible (so-called "garbage"). Solutions fall roughly into three main categories, each with pros and cons:

- ***Manual deallocation.*** Languages like C, C++, and Ada allow the programmer to free memory that the program has explicitly allocated. This "trust the programmer" philosophy runs the risk of storage leakage (if the programmer forgets to deallocate), dangling references (if the storage is deallocated while still being referenced), double freeing (deallocating the same storage more than once), and heap fragmentation (insufficient contiguous space for an allocation).

  Variations on this approach allow the programmer to define memory pools, where each allocation for the same pointer type uses storage in a specific fixed-size area. If all objects in a pool have the same size, this can prevent fragmentation and enable constant-time allocation and deallocation. Further, the entire pool for a locally declared pointer type can be reclaimed at once at scope exit.

  The manual deallocation approach is efficient but error-prone. Deciding when it is safe to free a pointer can entail complicated analysis, and predicting an appropriate size for a memory pool can likewise be difficult.

  Manual deallocation does not help if a data structure is implicitly allocated on the heap by the compiled code. In such cases, the compiler vendor's run-time support library is responsible for freeing the storage when it is safe to do so.

- ***Automatic deallocation.*** Languages like Java and Python do not allow the programmer to explicitly free storage but instead provide a run-time service known as a garbage collector to automatically deallocate inaccessible values. Garbage collection has been the subject of research since the 1960s and can be implemented through a wide range of techniques. Some are incremental, incurring an overhead in time and/or space that is distributed across program execution, while others use "stop the world" algorithms that kick in and reclaim garbage when the available heap space gets below a specific threshold.

  Garbage collection prevents deallocation-related bugs and avoids storage leakage from inaccessible values, and some techniques can also prevent heap fragmentation. However, algorithmic complexity as well as time and/or space expense have made garbage collection impractical for critical real-time software. The garbage collection approach is safe (assuming that it is implemented correctly) but incurs a penalty in performance and predictability.

- ***Semi-automatic deallocation.*** Complementing their support for manual deallocation, languages like C++ and Ada allow the programmer to define special functions, on a per-type basis, that are invoked at specific points in program execution where storage management may be affected. Common "hook" points are assignment, parameter passing, and scope exit. The functions can be defined to implement basic reclamation strategies such as reference counting.

  Semi-automatic deallocation can effectively support storage reclamation for non-cyclic data structures, with better time and space predictability than for general garbage collection. It also is preferable methodologically to manual deallocation, since it places the responsibility for resource cleanup on the implementer of the resource rather than on the user. The approach is safe and can be implemented efficiently but does not scale up to complex data structures.

The “billion dollar mistake”

A related issue is the semantics for uninitialized pointers. Languages typically provide a special “null” value as a solution: a pointer that does not reference any value. In some languages (e.g., Ada and Java), the null value is used as the default initialization for pointers, and attempting to dereference a null pointer raises a run-time exception. Other languages (e.g., C and C++) treat such attempts as undefined behavior. In either case, it’s a program error, and bugs related to null pointers have infected programs for decades. At the 2009 QCon conference in London, Prof. Anthony Hoare took responsibility for introducing null pointers in ALGOL W back in 1964 and thereby creating what he called “my billion-dollar mistake”. That number has since gone up considerably. Evidence points to (pardon the pun) a NULL dereference from C++ code, in a file incorrectly installed as part of a CrowdStrike software update, as the trigger for the Windows crash that crippled enterprise IT systems worldwide in late July 2024.

The pointer mole

All of this presents a dilemma for language designers, compiler writers, and software developers, especially in application areas demanding high assurance, run-time efficiency, and space/time predictability. In some situations, it’s preferable to avoid the problem rather than attempt a solution. As an example, coding standards for software that needs to be certified under rigorous assurance standards typically restrict dynamic allocation to the initialization/startup code and disallow deallocation. With these limitations, the software will not exhaust heap storage and will not suffer dangling references. However, the restrictions do not prevent concurrency-related pointer errors on accesses to shared data, and they also require analysis to ensure no dereferencing of null pointers.

There are no perfect solutions, and to a software developer it may seem like a frustrating game of “Whack-A-Mole”. If you want a pointer facility that is expressive, safe, and efficient, then pick two; the third will be elusive. Expressive and safe? Try Java, but be prepared to sacrifice performance. Expressive and efficient? Welcome to C and C++, but realize that safety requires painstaking effort and comes in spite of, rather than because of, pointer semantics. Safe and efficient? The SPARK subset of Ada fills the bill, but with a subset of typical pointer functionality. The full Ada language comes close to meeting all three goals, but its null pointers repeat Prof. Hoare’s billion-dollar mistake. The pesky pointer mole seems impervious to destruction.

The Rust approach

Pointers in the Rust language bring a fresh approach to mole whacking. Targeted to high-performance / high-assurance embedded systems, Rust supplies a pointer facility with the goal of jointly supporting safety and efficiency without sacrificing expressive power. The key is a distinction between high-level “safe pointers” and low-level “raw pointers”. Both can be implemented efficiently; the former come with assurance guarantees (including across threads) while the latter lack those guarantees. Raw pointers are potentially unsafe and thus require extra verification effort to ensure safety.

Safe pointers are based on several principles:

- The absence of null pointers
- An ownership concept that prohibits manual deallocation and enables automatic storage reclamation without a garbage collector
- A borrowing concept that allows multiple references to share the same value but ensures exclusivity for writing to (“mutating”) a referenced value

The rules for safe pointers make it possible to detect most pointer errors at compile time – no dangling references, reading uninitialized pointers, double-freeing, or corrupting shared-data – while facilitating code optimization and helping to prevent heap storage leakage without the overhead of a garbage collector.

Safe pointers come in two varieties:

- Pointers that can only point to values allocated on the heap. Rust's standard prelude supplies several types of heap-only pointers, including `String`, `Box<T>`, and `Vec<T>`. These are sometimes referred to as *smart pointers*.

  Representationally, a smart pointer is not just a pointer (address), it can contain supplementary data. For example, a variable of type `Vec<T>` is implemented as a struct comprising not simply the heap address for the start of the vector but also length and capacity fields.

- Pointers known as *references*, which can point to values on the heap, on the stack, or in static memory (including ROM). The type for such a pointer has the form `&T` or `&mut T` where `T` is a type; the former allows the reference to read from but not write to (mutate) the referenced value, whereas `&mut T` allows both reading and mutating.

In either case a pointer `p` can be dereferenced via the syntax `*p`.

Let's see how all of this works.

No null values

Rust avoids Prof. Hoare's "billion-dollar mistake" and uses flow-analysis based compile-time checks to ensure that safe pointers are initialized before being used. If the programmer needs an explicit way to simulate a null value, the predefined generic enum `Option<T>` does the job, where `T` is a safe pointer type:

- If the tag of an `Option<T>` value is `Some`, then a pointer of type `T` is present.
- If the tag is `None`, then there is no associated pointer value.

An `Option<T>` value cannot be used directly as a value of type `T`. Instead, it has to be queried, typically in a match statement, with code that only accesses the value when the tag is `Some`. Misuses are caught at compile time. Here's an example.

```
let v : [Option<Box<i32>>; 2] =
            [ Option::None,
              Option::Some(Box::new(100)) ];
 // This code is correct:
 for item in &v {
    match item{
       None    => println!("Nothing here"),
       Some(p) => println!("Boxed value: {}", *p),
    }
 }

 // The let statement below is illegal:
 // No implicit cast from Option<T> to T
let ptr : Box<i32> = v[1]; // Illegal: type mismatch
```

Here v is an array of two `Option<Box<i32>>` `values: None`, and a `Some` variant that contains a `Box<i32>` pointer to a heap value set to 100. Processing an `Option` involves interrogating the "tag", as is done in the match statement. It's a compile-time error to attempt to use an `Option` as a value of the type of its `Some` variant.

Ownership, dynamic allocation, and dropping

As noted above, Rust supplies standard pointer types like `Box<T>` that can only be used for dynamic allocation. A pointer from one of these types, unless uninitialized, *owns* the value that it points to, and, aside from reference-counting types that will be described below, the owning pointer is unique: no other pointers can share ownership. Assignment, including in implicit contexts such as parameter passing and field initialization, transfers ownership of the value from the source to the target. In Rust parlance, the pointer is *moved* from the source to the target. Except for the special "no-op" case of self-assignment where the source and target pointers are the same, the move for single-ownership pointers involves several steps:

- If the target is a valid (initialized) pointer, the Rust run-time implementation automatically reclaims ("drops") the heap value that the target references (and recursively if the value itself contains single-ownership pointers to heap values).
- The source pointer is copied to the target.
- The source is treated as uninitialized; subsequent attempts to dereference it before it is reinitialized will be flagged as compile-time errors.

At the end of the scope containing the declaration of an initialized single-ownership pointer variable, the Rust run-time implementation automatically drops the referenced heap value (and recursively if the value itself contains pointers owning heap values). The pointer variable is owned by its containing scope and is dropped implicitly when the scope's stack frame is popped. Note that the term "pointer variable" also refers to pointers that occur as formal parameters, struct fields, array and vector elements, etc.

Rust's single ownership approach reclaims inaccessible storage without needing a garbage collector, prevents dangling references, and also avoids data corruption of heap values (if a heap object can only have one owner, it cannot be owned by a pointer in another thread). It facilitates the implementation of types like Vec<T>, whose values require reallocation when their capacity is exceeded.

However, by itself, the single-ownership model is too restrictive:

- Transferring ownership each time a pointer is passed to a function leads to an awkward style if the pointer needs to be used after the function returns.
- The single-ownership restriction inhibits the implementation of some common data structures and does not support the use of pointers for indirection (i.e., pointers to declared variables rather than to dynamically allocated values).

The following elements of Rust's safe pointer facility address these limitations.

Indirection, references, and the Borrow Checker

Arguably the most novel aspect of Rust, and the feature that is the most challenging for new users, is the treatment of pointers that can be used for indirection. Known as references, these pointers have a C-like syntax but with a few new wrinkles:

- `&x` is an i*mmutable borrow* of x. It creates a reference to a mutable or immutable value x, through which x can be read but not mutated. Ownership of x is not affected. The "&" operator can be applied to any construct that has a run-time presence, including declared variables, heap values, literals, and function names.
- `&mut x` is a *mutable borrow* of x. It is analogous to `&x` but allows mutating x and cannot be applied to immutable values.
- `&T` is the type for `&x` values where x is of type `T`; i.e., for pointers that can reference values of type `T` but do not mutate them.
- `&mut T` is an analogous type for `&mut x` values, which also allows mutating the referenced values.

In less safe languages such a facility would risk dangling references or data corruption, while also inhibiting some useful optimizations. Through a compiler functionality known as the Borrow Checker, Rust enforces restrictions that avoid these drawbacks.

Here's an example that shows how Rust prevents dangling references:

```
{                              // outer scope
   let ptr : &i32;             // (1) ptr is owned by the outer scope
   {                           // inner scope
      let n : i32 = 100;       // n is owned by the inner scope
     ptr = &n;                // (2) ptr borrows an immutable reference to n
      println!("{}", *ptr);   // (3) OK to dereference ptr
   }                           // (4) n dropped at end of inner scope
   println!("{}", *ptr);       // (5) Illegal
}                              // (6) ptr dropped at end of outer scope
```

Line (1) declares `ptr` as an immutable reference to an `i32` value, and line (2) initializes `ptr` with a reference to n (an immutable borrow; the owner of n is its enclosing scope). This is a potential dangling reference but is not dangerous yet, and indeed the use of `*ptr` at (3) is legal. However, an attempt to use `*ptr` at line (5) would be an actual dangling reference since n would have been dropped at the end of the inner scope (4). The Borrow Checker detects this error through lifetime analysis and rejects the program.

In the absence of line (5), the lifetime of `ptr` would only extend from line (1) through its last use (line (3)), even though it does not get dropped until exit from the scope at (6). However, in the presence of line (5) the lifetime of `ptr` extends through (4), which is longer than the lifetime of its referent n, and thus the program is illegal. The fact that a variable's lifetime may be shorter than its lexical scope (a property known as "non-lexical lifetimes") provides flexibility without compromising safety.

An immutable variable can only be borrowed immutably. A mutable variable x can be borrowed either mutably via `&mut x`, or immutably via `&x`; in the latter case x can be mutated but not through the `&x` pointer.

A linchpin of Rust pointer safety is exclusivity of mutable borrows. While a variable is borrowed immutably, all other borrows (whether mutable or immutable) are prohibited. On the other hand, simultaneous immutable borrows are permitted. Direct uses of variables are considered borrows.

In a multithreaded environment this property is commonly known as "Concurrent Read, Exclusive Write", or CREW. Rust enforces this principle, while also preventing dangling references, through a variety of features including read-write locks and mutexes. Direct use of shared data across threads is permitted – static data can be referenced in functions and closures that are passed to `thread::spawn()` – but is discouraged for mutable values. There are no language-provided checks that verify exclusivity of mutable borrows of static variables, and to make this clear in the source program all uses of mutable static variables must be within special syntax ("unsafe" blocks). The programmer is responsible for ensuring the absence of interference. Immutable static variables may be safely shared, however.

Aside from mutable static variables, exclusivity for mutable borrows is enforced within a single thread. This prevents some error-prone constructs while enabling useful optimizations. Here's an example; the cache function is from Jon Gjengset's book *Rust for Rustaceans*:

```
fn cache(input: &i32, sum: &mut i32){
   *sum = *input + *input;
   assert_eq!(*sum, 2 * *input)
}

fn main(){
   let m     : i32 = 100;
   let mut n : i32 = 0;
   cache(&m, &mut n);    // (1) OK
   println!("{m}, {n}"); // Prints 100, 200
   cache(&n, &mut n);    // (2) Illegal
}
```

The Borrow Checker allows the invocation of (1), since m and n are distinct variables, but rejects the code at (2) which attempts to borrow n both immutably and mutably. The effect is that the compiler can safely cache the value of `*input` on entry to the function, since `input` and `sum` cannot reference the same variable. Compare this to the C function shown earlier in this article, where the optimization would have changed the effect of the program.

As an aside, the ownership and borrowing concepts that underlie pointer safety guarantees are not unique to Rust. The SPARK language has somewhat similar notions, but with restrictions on pointers (known as "access values" in SPARK) that allow formal verification of program properties.

Reference-counting pointers

Although single ownership simplifies storage management, the limit of one owner per value can be overly constraining. For example, in a directed acyclic graph several nodes may point to the same node, but there is no clear owner; the referenced node should be reclaimed when and only when its last extant owner is dropped.

This scenario may sound familiar: reclamation can be managed by the classical reference count technique and is realized in Rust through the generic smart pointer types `Rc<T>` and `Arc<T>` ("A" is for "Atomic"). These are analogous to `Box<T>` in that they support dynamically allocating values of `type T` on the heap, but for `Rc<T>` and `Arc<T>` the heap storage for the `T` value includes a count of the number of owners. The reference count management for `Rc<T>` is not thread safe, so `Rc<T>` can only be used in single-threaded code. `Arc<T>` has the necessary protection, at some cost in performance, and is safe in multi-threaded code.

Assignment of reference counting pointers has standard Rust "move" semantics, with ownership transferred from source to target. The only new wrinkle is that, rather than automatically reclaiming the heap storage for the value referenced by the target pointer, the Rust implementation subtracts 1 from the reference count for this value. When the count goes to 0, the storage is reclaimed.

To share ownership of a value that is referenced by a reference-counting pointer, use the `clone()` function from `Rc<T>` or `Arc<T>` which, despite its name, does not allocate a new copy of the heap value but instead copies the pointer:

```
let p : Rc<i32> = Rc::new(100);
let q : Rc<i32> = Rc::new(200);
q = Rc::clone(&p);
```

The assignment statement comprises three actions:

- `Rc::clone(&p)` simply returns the pointer `p` and increments the reference count for `*p` by `1`.
- The reference count for `*q` is decremented by 1, and, since it is now `0`, the storage for `*q` is reclaimed.
- The `Rc::clone()` result `(pointer p)` is copied to `q`, resulting in shared ownership of `*p` by `p` and `q`.

The reference count (more strictly, the count of strong references) for a value referenced by an `Rc<T>` or `Arc<T>` pointer `p` can be interrogated by the function `Rc::strong_count(&p)` or `Arc::strong_count(&p)`, respectively.

To preserve the Rust principle that a value cannot be mutated through one pointer while being read or mutated by another, the referents of both `Rc<T>` and `Arc<T>` pointers must be immutable. This is a significant restriction, but, as will be shown below, Rust provides an escape.

Here is an example of multiple ownership through reference-counting pointers; the illegal lines are commented out.

```
use std::rc::Rc; // std crate, rc module, Rc type
use std::ptr;

fn main(){
   let ptr1 : Rc<i32> = Rc::new(100);
// *ptr1 += 1;     // Illegal, reference counted values are immutable
   {
      let ptr2 = Rc::clone(&ptr1);     // ptr1 and ptr2 share ownership
     assert!(!ptr::eq(&ptr1, &ptr2)); // ptr1, ptr2 are different variables
      assert_eq!( ptr1, ptr2 );       // But they contain the same pointer
      assert_eq!( &*ptr1, &*ptr2 );    // Equivalent to previous assertion
      assert_eq!( *ptr1, *ptr2 );      // The referenced values are equal
      let ptr3 = ptr2;                 // Move semantics
      // ptr1 and ptr3 share ownership, ptr2 uninitialized
      assert_eq!( Rc::strong_count(&ptr3), 2 );
//    println!("{}", *ptr2);   // Illegal, since ptr2 is uninitialized
   }                           // Drop ptr3, decrement reference count
   assert_eq!( Rc::strong_count(&ptr1), 1 );
```

Here is an example of multiple ownership through reference-counting pointers; the illegal lines are commented out.

```
use std::rc::Rc; // std crate, rc module, Rc type
use std::ptr;

fn main(){
   let ptr1 : Rc<i32> = Rc::new(100);
// *ptr1 += 1;     // Illegal, reference counted values are immutable
   {
      let ptr2 = Rc::clone(&ptr1);     // ptr1 and ptr2 share ownership
     assert!(!ptr::eq(&ptr1, &ptr2)); // ptr1, ptr2 are different variables
      assert_eq!( ptr1, ptr2 );       // But they contain the same pointer
      assert_eq!( &*ptr1, &*ptr2 );    // Equivalent to previous assertion
      assert_eq!( *ptr1, *ptr2 );      // The referenced values are equal
      let ptr3 = ptr2;                 // Move semantics
      // ptr1 and ptr3 share ownership, ptr2 uninitialized
      assert_eq!( Rc::strong_count(&ptr3), 2 );
//    println!("{}", *ptr2);   // Illegal, since ptr2 is uninitialized
   }                           // Drop ptr3, decrement reference count
   assert_eq!( Rc::strong_count(&ptr1), 1 );
}
```

Interior mutability

The immutability requirement for reference-counted values is well motivated but overly restrictive, and Rust provides an explicit mechanism for arranging mutability within a value that is otherwise immutable. Safety is preserved, since a check enforces exclusivity of mutable borrows, but it is performed at run time; a failed check results in a `panic` that terminates the affected thread.

The feature that allows mutability within an otherwise immutable value is known as *interior mutability* and is realized through the `Cell<T>` and `RefCell<T>` types. `Cell<T>` has `get()` and `set()` methods that are appropriate if `T` has copyable (versus movable) values; for example, scalar types such as i32. Otherwise `RefCell<T>` should be used, which comes with `borrow()` and `borrow_mut()` methods.

The interior mutability property applies more generally, for example to define a struct type where some fields are immutable and others are mutable, but it is especially useful for reference-counted values.

Here is an example with a borrowing violation that is detected at run time:

```
use std::rc::Rc; // std crate, rc module, Rc type
use std::cell::{RefCell, RefMut, Ref};

fn main(){
   let ptr1 : Rc<RefCell<i32>> = Rc::new(RefCell::new(100));
  // ptr1 is a pointer to a mutable reference-counted i32 value on the heap
   {
      let ptr2           : Rc<RefCell<i32>> = Rc::clone(&ptr1);  //(1)
      let mut borrow_mut : RefMut<i32>      = ptr2.borrow_mut(); //(2)
      *borrow_mut += 1;
```

```
            assert_eq!( Rc::strong_count(&ptr1), 2 );
        let borrow_immut   : Ref<i32>         = ptr1.borrow();     // (3) panic
        assert_eq!( *borrow_immut, 101 );
    }
    assert_eq!( Rc::strong_count(&ptr1), 1 );
}
```

At line 1, `ptr1` and `ptr2` point to and share ownership of the `RefCell value 100`, which has been allocated on the heap and has a reference count of 2. At line 2, this value is borrowed mutably. The statement at line 3 attempts to borrow this same value; this is a run-time error `(panic)` causing program termination. The correction is to ensure that `borrow_mut` is dropped before attempting the second borrow:

```
use std::rc::Rc; // std crate, rc module, Rc type
use std::cell::{RefCell, RefMut, Ref};

fn main(){
   let ptr1 : Rc<RefCell<i32>> = Rc::new(RefCell::new(100));
   {
      let ptr2             : Rc<RefCell<i32>> = Rc::clone(&ptr1);  // (1)
      let mut borrow_mut   : RefMut<i32>      = ptr2.borrow_mut(); // (2)
      *borrow_mut += 1;
      assert_eq!( Rc::strong_count(&ptr1), 2 );
   }                                                               // (3) OK
   let borrow_immut        :    Ref<i32>      = ptr1.borrow();     // (4)
   assert_eq!( *borrow_immut, 101 );
   assert_eq!( Rc::strong_count(&ptr1), 1 );
}
```

This program starts the same way as the previous version, sharing ownership of the `RefCell` at line (1) and mutably borrowing the value at line (2). But the scope of the mutable borrow `(borrow_mut)` ends at (3), and thus the subsequent borrow at line (4) is permitted. The program executes with no run-time errors.

Weak references

Complex data structures may have cycles, with separately allocated nodes that reference each other either directly or indirectly. Cycles present a challenge for reference counting, since the interdependence between nodes prevents the nodes' reference counts from reaching zero. To support the definition of cyclic data structures, Rust differentiates between two categories of referencing-counting pointers:

- A "strong" reference determines storage reclamation and is the default for `Rc` or `Arc` pointers. A heap value referenced by an `Rc` or `Arc` pointer is reclaimed when its strong reference count is decremented to zero.
- A "weak" reference does not affect storage reclamation. Its referent can be reclaimed even when its count of weak references is non-zero, provided that the count of strong references is 0.

A strong reference can be downgraded to a weak reference, and in the other direction, a weak reference can be upgraded to strong.

Weak references are useful when the nodes in a data structure exhibit a natural “parent-child” relationship. The parent has a strong reference to a child, and the child has a weak reference to a parent. One example is a doubly-linked list where each node contains a `data` field (say an `i32`) and two pointers (actually `Option` values):

- a next pointer which is `None` for the tail node and otherwise a `Some` variant that references the successor node, and
- a prev pointer which is `None` for the head node and otherwise a `Some` variant that references the predecessor node.

Although cycles exist through the combination of prev and next links, we can ensure proper storage reclamation by defining the next links as strong references and the prev links as weak. The implementation of the methods provided for the data structure ensure that the next links themselves do not create cycles of strong references.

Here is a Rust definition for the relevant data structures and a sample implementation of several methods:

```
use std::cell::RefCell;
use std::rc::{Rc, Weak};

struct Node { data: i32,
              next: Option<Rc<RefCell<Node>>>,
              prev: Option<Weak<RefCell<Node>>>, }

struct DoublyLinkedList { head: Option<Rc<RefCell<Node>>>, }

impl DoublyLinkedList {
    fn new() -> Self {
        Self { head: None }
    }

    fn prepend(&mut self, value: i32) {
        let new_node: Rc<RefCell<Node>> = Rc::new(RefCell::new(Node {
            data: value,
            next: self.head.clone(),
            prev: None,
        }));

        if let Some(old_head) = self.head.take() {
            old_head.borrow_mut().prev = Some(Rc::downgrade(&new_node));
            // prev is now a weak reference
        }

        self.head = Some(new_node);
    }
}

fn main() {
    // Create a list with two nodes
  let mut list: DoublyLinkedList = DoublyLinkedList::new();
```

```
    list.prepend(10); // first node
    list.prepend(20); // second node

    // Get a strong reference to the head of the list
    let ptr1: Rc<RefCell<Node>> = list.head.clone().unwrap();

    assert!(Rc::strong_count(&ptr1) == 2);
    assert!(Rc::weak_count(&ptr1) == 1);

    // Enter an inner block, add references, and prepend a third node
    {
        // Temporary strong and weak references to ptr1
        let _temp_strong = Rc::clone(&ptr1);
        let _temp_weak = Rc::downgrade(&ptr1);

        list.prepend(30); // third node at the head
        println!("Inside inner block after prepending a third node:");

        assert!(Rc::strong_count(&ptr1) == 3);
        assert!(Rc::weak_count(&ptr1) == 2);

        // Display list contents: 30 20 10
        let mut current = list.head.clone();
        while let Some(node) = current {
            print!("{} ", node.borrow().data);
            current = node.borrow().next.clone();
        }
    }

    // Exit inner block, which decrements counts again

    assert!(Rc::strong_count(&ptr1) == 2);
    assert!(Rc::weak_count(&ptr1) == 1);
}
```

An important (implicit) property of the doubly linked list is the absence of cycles of strong references. If application code creates a doubly linked list where some node contains a next reference to itself or to a previous node, then the strong reference count will never get to zero. To prevent storage leakage two approaches are available:

- Explicitly break the cycle by setting to None the next link of one of the nodes in the cycle; this will allow automatic reclamation.
- Define the data structure with raw pointers (see below) and use manual deallocation.

Raw pointers

For low-level programming and as the implementation basis for safe pointers and custom memory management, Rust provides a facility known as raw pointers. These provide C-like functionality, but also C-like lack of checking. Raw pointers come in two varieties:

- `*const T (immutable)`, which allow reading from but not assigning to the dereferenced T value, and
- `*mut T (mutable)`, which allow both reading from and writing to the dereferenced T value

Raw pointers can be created either through normal references (using the "&" operator) or dynamic allocation, and they can be converted (cast) to and from safe pointers. However, a raw pointer can only be dereferenced in an unsafe block. If the alloc() function in the std::alloc module is used in a function or block to allocate storage for a raw pointer, then the program needs to deallocate the storage by calling the dealloc() function explicitly.

Here is an example:

```
fn main() {
    let x     : i32 = 5;
    let mut y : i32 = 10;

    let mut raw_ptr1: *const i32 = &x;
    let raw_ptr2    : *mut i32   = &mut y;

    unsafe{
      assert_eq!(*raw_ptr1, 5);
//    raw_ptr2 = raw_ptr1; // (1) Illegal, mutability mismatch
    }

    unsafe {
        raw_ptr1 = &y as *const i32;
        raw_ptr1 = raw_ptr2; // Equivalent to previous line
//      *raw_ptr1 = 1;   // (2) Illegal, since raw_ptr1 is *const

        *raw_ptr2 += 1;
        assert_eq!(*raw_ptr1, 11);
    }

    // Cast from safe pointer to raw pointer
    let safe_ptr  : Box<i32> = Box::new(100);
    let raw_ptr3  : *const i32 = &*safe_ptr;

    unsafe {
        assert_eq!(*raw_ptr3, 100);
    }
}
```

Raw pointers are efficient and are not necessarily unsafe; as shown in the example (commented lines (1) and (2)), Rust provides some compile-time checks that prevent unsafe practices. However, in the absence of the guarantees that come with safe pointers, the developer will need to work harder to verify the code; raw pointers enable all of the traditional errors (dangling references, storage leakage, double freeing, null dereferencing, etc.).

An important application of raw pointers is as a toolkit for defining memory pools and custom allocation and deallocation mechanisms; these are especially useful in embedded applications. Examples are the `typed-arena`, `slab and mempool` crates in the Rust community's `crates.io` registry.

Raw pointers should not be (mis)used as a workaround for avoiding language-enforced checks. They serve a useful purpose as a means to interface with foreign (non-Rust) code and to implement low-level functionality.

Revisiting the mole

Has Rust vanquished the pointer mole? Can Rust programmers write safe and efficient code while comfortably defining the underlying data structures? The answer: a slightly qualified yes. On the positive side, Rust detects many pointer errors at compile time, it reclaims storage automatically without the overhead of a general garbage collector, and its type definition facility offers general-purpose functionality. However, and not surprisingly in light of the tensions and tradeoffs intrinsic to any pointer facility, Rust's approach does have some drawbacks.

Arguably the most significant issue is the learning curve most Rust programmers will need to navigate in order to become conversant with the language's pointer semantics. The Borrow Checker is an algorithm based on source code path analysis. Although its essence can be distilled into a memorable mantra – values can be borrowed mutably and exclusively or else immutably and shared – specific applications of the rules may cause surprises and, as can be seen in the doubly-linked list example above, traditional data structuring idioms sometimes require a non-traditional style.

Here are two examples that are methodologically equivalent; each mutates a variable through a borrowed reference. But one is legal, and the other not. First the illegal code:

```
fn main(){
   let mut n = 100;
   println!(“n: {n}”);
   let p = &mut n;          (1)
   *p += 1;
   println!(“n: {n}”);      (2) Illegal
   println!(“*p: {}”, *p); (3)
}
```

At statement (1), `p` borrows `n` mutably; the lifetime of `p` (and the associated extent of `p's` mutual borrow of `n`) go through line (3). However, the use of `n` in the `println!()` macro at line (2) is an implicit immutable borrow of `n`. Since this occurs during the extent of the mutable borrow, the code is illegal. On the other hand, interchanging the last two lines makes the code legal:

```
fn main(){
   let mut n = 100;
   println!(“n: {n}”);
```

```
    let p = &mut n;         (1)
    *p += 1;
    println!("*p: {}", *p); (2) OK
    println!("n: {n}");     (3)
}
```

The difference is that now the lifetime of p ends at line (2), so the immutable borrow of n at line (3) does not occur during the extent of the mutable borrow.

The distinction between the two examples might not be immediately apparent. Aliasing in which a variable is modified indirectly and referenced directly may or may not be legal. Further, when interior mutability is required, and the borrow checking rules are enforced at run-time, complex analysis may be needed to guarantee the absence of borrow-related `panic`.

Another potential issue is the cost of reclamation when complex data structures are dropped. Despite the dangers of explicit deallocation in languages like C, C++, and Ada, the deallocation is apparent in the source code and, especially if memory pools with fixed-size blocks are used, its run-time cost can be predicted. In Rust, the deallocation is implicit, and careful analysis is needed to compute the run-time cost. Rust's custom storage pool mechanism can mitigate this problem, but it brings manual deallocation and its associated risks.

Notwithstanding these issues, Rust provides an expressive pointer facility that has advanced the state of the art and practice in helping programmers write safe and efficient code. Rust requires more ramp-up time and up-front effort than other languages, but by detecting most pointer errors at compile time it facilitates memory safety, avoids Prof. Hoare's billion-dollar mistake, and reduces verification costs. The frustrating pointer mole has not been completely whacked, but Rust has largely stunned it into submission.

About the author:

Dr. Benjamin Brosgol is a member of the senior technical staff at AdaCore. Throughout his career, he has focused on programming language technology for high-assurance software. He was a member of the design team for Ada 95 and also a member of the expert group that developed the Real-Time Specification for Java. Dr. Brosgol has delivered papers and presented tutorials on safety and security standards (DO-178C, Common Criteria) and programming languages (Ada, Java, C#, Python). He is an AdaCore representative on the FACE® Consortium (Future Airborne Capability Environment™) and has served as Vice Chair of that organization's Technical Working Group.

Dr. Brosgol holds a BA in Mathematics from Amherst College, and MS and PhD degrees in Applied Mathematics from Harvard University.