AdaCore TECH PAPER

MISSION CRITICAL RUST: Managing Memory

ミッションクリティカルシステムにおける Rust のメモリ管理

概要

ポインタに関するエラーは、何十年にもわたってソフトウェア開発者を悩ませてきました。Rust の革新的なアプローチは、ポインタを安全かつ効率的に扱えるようにします。

ポインタはコンピューティングの黎明期からプログラミング言語の基本要素であり、次の 2 つの目的で使われてきました。

- 間接参照（Indirection）：データ値をプログラム内でコピーせずに共有する手段。これは暗黙的（たとえば「参照渡し」のパラメータ）でも、明示的（たとえば C における「&x」や「*p」などの構文）でも可能です。

- 動的メモリ割り当て（Dynamic Allocation）：実行時にサイズが増減するデータ構造（リンクリスト、ツリー、グラフなど）を構築・操作するための手段。

ただ、強力な機能にはリスクも伴い、ポインタは、安全性やパフォーマンスを損なう要因になり得ます。本稿では、ポインタに起因する課題を掘り下げ、Rust がどのように対処しているかを解説します。Rust は、ポインタの取り扱いにおいて従来の言語とは異なる設計思想を採用しています。その結果、メモリの安全性が向上し、ポインタ関連のバグを早期に発見できるだけでなく、汎用的なデータ構造を表現する柔軟性と高いパフォーマンスも両立しています。

間接参照に関する課題

ポインタが、すでに破棄された値を参照し続けてしまうと、「ダングリング参照（dangling reference）」と呼ばれる厄介なバグが起こります。たとえば、C 言語で関数内のローカル変数への参照を返してしまうと、関数のスコープを抜けたあとにその参照を使うことになり、結果は未定義です。しかもこの手のバグは、問題が起きた瞬間ではなく、その無効な値を実際に使ったときに表面化するので、原因の特定がとても難しくなります。

複数のスレッドで同じ値を共有していて、それに対して適切な保護（排他制御）なしにアクセスすると、「データ競合（data race）」と呼ばれる深刻な問題が発生します。たとえば、あるスレッドが値を書き込んでいる最中に、別のスレッドがその同じ値を読み込んだり上書きしたりすると、値が壊れる可能性があります。
この問題も、ダングリング参照と同様に、バグが起きた瞬間には気づかず、後になってからおかしな動作として現れることが多いため、原因の特定が非常に難しくなります。並行プログラミングは実行の流れが一つではないため、それだけでもデバッグが難しいのに、データ競合があるとさらに状況が複雑になります。

また、間接参照は最適化の妨げになり、パフォーマンスに悪影響を与える可能性があります。

以下の C プログラムに見られるエイリアス（別名）問題はその例です：

```
#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
}
```

最適化コンパイラは、関数 func() に入った時点で *p をレジスタに保持し、return 文で再利用したいと考えます。しかし、*p と *q が重複しない、という本来は保証されない前提を置かなければならず、それは無効な仮定になってしまいます。

動的メモリ割り当てに関する課題

多くのアプリケーションでは、サイズが実行時に増えたり減ったりするデータ構造が必要になります。こうした場合、あらかじめ最大サイズを決めておくことが難しいため、プログラミング言語では「ヒープ」と呼ばれる領域を使って、動的にメモリを確保する仕組みが用意されています。

ただし、問題になるのが「使われなくなったメモリ（いわゆる“ガベージ”）」を、いつ、どうやって解放するかという点です。この課題の解決策は、大きく分けて 3 つの方法があり、それぞれにメリットとデメリットがあります。

- **手動によるメモリ解放**
  C、C++、Ada 言語では、プログラマが明示的に確保したメモリを解放することができます。この「プログラマを信頼する」という思想には、次のようなリスクがあります。プログラマが解放を忘れた場合のストレージリーク（記憶領域の消失）、まだ参照中のストレージを解放してしまうことによるダングリング参照、同じストレージを複数回解放する二重解放、そして確保のために十分な連続領域がなくなるヒープ断片化などです。

  この方式のバリエーションとして、プログラマがメモリプールを定義できるものがあります。同じポインタ型に対する各メモリ確保を、特定の固定サイズ領域内のストレージで行う方法です。プール内のすべてのオブジェクトが同じサイズであれば、断片化を防ぎ、確保や解放を一定時間で実行できるようになります。さらに、ローカルに宣言されたポインタ型に対するプール全体を、スコープの終了時にまとめて解放することも可能です。

  手動によるメモリ解放の方式は効率的ですが、エラーが発生しやすい方法です。ポインタを安全に解放できるタイミングを判断するには複雑な解析が必要になる場合があり、メモリプールの適切なサイズを予測するのも難しいことがあります。

さらに、コンパイルされたコードによってデータ構造が暗黙的にヒープ上に確保される場合、手動解放は役に立ちません。この場合には、コンパイラが提供するランタイムサポートライブラリが、安全に解放できるタイミングでストレージを解放する責任を負います。

- **自動解放（Automatic deallocation）**
  Java や Python などの言語では、プログラマが記憶領域を明示的に解放することはできません。その代わり、ランタイムサービスとして ガベージコレクタ（GC）を用いて、アクセス不能な値を自動的に解放します。
  ガベージコレクションは 1960 年代から研究が続けられており、多様な技術が存在します。一部は「インクリメンタル型」と呼ばれ、実行全体に渡って少しずつ処理負荷を分散させる方式であり、他には「ストップ・ザ・ワールド型」があり、ヒープの空き容量が一定以下になるとプログラムの実行を一時停止して一括回収を行います。

  ガベージコレクションは、解放に関連するバグを防ぎ、到達不能な値によるメモリリークを回避できます。一部の手法ではヒープの断片化も防げます。しかし、アルゴリズムの複雑さや処理の時間・空間コストが、リアルタイム性が要求されるソフトウェアには不適であることが多いのが現状です。ガベージコレクション方式は安全性を提供します（正しく実装されていれば）が、パフォーマンスと予測可能性において代償が発生します。

- **準自動解放（Semi-automatic deallocation）**
  C++や Ada 言語では、手動によるメモリ解放に加えて、型ごとに特別な関数（いわゆるカスタムハンドラ）を定義することができます。この関数は、メモリの割り当て時やスコープを抜けるときなど、特定のイベントに応じて自動的に呼び出され、メモリ管理の挙動をカスタマイズできます。

  タイミングとしては、代入時、関数へのパラメータ渡し時、スコープ終了時などがあり、こうしたポイントを「フック」して処理を追加できます。たとえば、参照カウントを使ってオブジェクトの寿命を自動的に管理する、といった仕組みもこの方法で実装可能です。

  この方法は、循環参照のないデータ構造に対しては、記憶領域の解放を効果的に行うことができます。一般的なガベージコレクション（GC）と比べても、実行時間やメモリ使用量の予測がしやすく、リアルタイム性が求められる場面にも向いています。また、手動解放と違って、リソースの管理責任を使う側ではなく実装側に持たせる設計ができるため、ソフトウェア全体の堅牢性も高まります。この方式は、安全かつ効率的に実装できますが、複雑で相互参照を含むようなデータ構造には対応が難しいという課題もあります。

10 億ドルの失敗

もうひとつ関連する重要な問題が、初期化されていないポインタの扱い（セマンティクス）です。多くのプログラミング言語では、この問題に対処するために「null」という特別な値を導入しています。null は「どのオブジェクトも指していない」ということを示すポインタで、初期化されていない状態を明示的に表す手段として使われています。

一部の言語（たとえば Ada や Java）では、null 値がポインタのデフォルト初期値として使われ、null ポインタを参照しようとすると実行時に例外が発生します。
一方、C や C++ のような言語では、そのような操作は「未定義の動作」として扱われます。いずれの場合であっても、それはプログラム上の誤りであり、null ポインタに関連するバグは何十年にもわたってソフトウェアを悩ませてきました。

2009 年にロンドンで開催された QCon カンファレンスで、アンソニー・ホーア（Anthony Hoare）教授は 1964 年に ALGOL W に null ポインタを導入したことを認め、それを「私の 10 億ドルの失敗（billion-dollar mistake）」と呼びました。その被害額はその後さらに大きくなっています。実際、2024 年 7 月下旬に世界中の企業 IT システムを麻痺させた Windows の大規模障害も、CrowdStrike のソフトウェアアップデートで誤ってインストールされたファイル内の C++ コードにおける NULL 参照が引き金だったと見られています。

ポインタ・モグラ叩き

こうした状況は、言語設計者やコンパイラ開発者、ソフトウェア開発者にとって大きなジレンマを生みます。特に、高い信頼性や実行時効率、メモリや時間の予測可能性が求められる分野では顕著です。
場合によっては、問題の解決を試みるよりも、そもそも問題を避けるほうが望ましいこともあります。
例えば、厳格な信頼性基準に基づく認証が必要なソフトウェアでは、コーディング標準として動的メモリの割り当てを初期化や起動処理のみに限定し、解放は行わないことが定められることがあります。このような制限を設ければ、ヒープ領域を使い果たすことやダングリング参照の発生を防ぐことができます。
しかし、これらの制限だけでは、共有データへのアクセス時に起こる並行性に関するポインタの誤りは防げません。また、null ポインタの参照が発生しないことを確認するための分析も必要になります。

完璧な解決策は存在せず、ソフトウェア開発者にとっては、まるで「モグラ叩き」のようなもどかしい状況に感じられるかもしれません。
表現力があり、安全で、なおかつ効率的なポインタ機構を求めるなら、三つのうち二つを選ぶことはできますが、残りの一つは手に入りにくいという状況です。

- 表現力と安全性を重視するなら Java が候補ですが、その分パフォーマンスは犠牲になります。
- 表現力と効率を重視するなら C や C++ が適していますが、安全性を確保するには非常に手間がかかり、ポインタの意味論のおかげで安全になるわけではありません。
- 安全性と効率を重視するなら Ada の SPARK サブセットが役立ちますが、通常のポインタ機能は一部しか利用できません。

Ada 言語は、この 3 つの目標をほぼ満たしていますが、null ポインタに関してはホーア教授の「10 億ドルの失敗」を繰り返しています。
厄介なポインタというモグラは、どうやら完全には退治できないようです。

Rust のアプローチ

Rust 言語のポインタは、モグラたたきに新しいアプローチをもたらします。高性能かつ高信頼性が求められる組込システムを想定して設計された Rust では、安全性と効率を両立しつつ、表現力も犠牲にしないポインタ機構を提供しています。

ポイントは、高レベルの「安全なポインタ（safe pointer）」と、低レベルの「生ポインタ（raw pointer）」を区別している点です。どちらも効率的に実装できますが、安全なポインタは保証（スレッド間を含む）付きであり、生ポインタはそのような保証がありません。生ポインタは潜在的に安全でないため、使用する際には追加の検証作業が必要になります。

安全なポインタ（safe pointers）は、以下の原則に基づいて設計されています：

- null ポインタの排除
- 所有権の概念：手動によるメモリ解放を禁止し、ガベージコレクタを用いることなく自動的なメモリ解放を可能にします
- 借用の概念：複数の参照が同一の値を共有することを許容する一方で、値の書き換え「変更」に関しては排他性を確保します

安全なポインタに関する規則により、ほとんどのポインタ関連のエラー「たとえばダングリング参照（解放済み領域へのアクセス）、未初期化ポインタの読み出し、二重解放、共有データの破壊など」をコンパイル時に検出できます。これにより、コード最適化が容易になるとともに、ガベージコレクタを用いることなくヒープ領域のメモリリーク防止にも寄与します。

安全なポインタには、主に次の 2 種類があります：

- ヒープ上に確保された値のみを指すポインタ
  Rust の標準プレリュード（prelude）には、String、Box<T>、Vec<T> など、ヒープ専用ポインタ型がいくつか含まれています。これは「スマートポインタ（smart pointers）」と呼ばれます。

  表現上、スマートポインタは単なるポインタ（アドレス）ではなく、補助的なデータを含む構造です。たとえば、Vec<T> 型の変数は、ヒープ上のベクタの開始アドレスに加えて、長さ（length）および容量（capacity）の情報を含む構造体として実装されています。

- 参照（reference）と呼ばれるポインタ
  これは、ヒープ領域、スタック領域、または静的メモリ（ROM を含む）上の値を指すことができます。型としては &T または &mut T の形式をとり、&T は読み取り専用の参照、&mut T は読み書き可能な可変参照です。

いずれの場合も、ポインタ p は構文 *p を使って参照先の値の取得ができます。では、これらがどのように機能するのかを見ていきましょう。

Null 値を使用しない設計

Rust は ホーア（Hoare） 教授が指摘した「10 億ドルの失敗」（null ポインタによるエラー）を回避するために、フロー解析に基づいたコンパイル時チェックを行い、安全なポインタが必ず利用前に初期化されるようにしています。
もしプログラマが明示的に「値が存在しない」状態を表現したい場合には、あらかじめ用意されている汎用的な列挙型 **Option<T>** を使います。ここで T は安全なポインタ型です。

- **Option<T>** のタグが **Some** なら、その中に型 **T** のポインタが格納されています。
- **タグが None** の場合は、関連するポインタ値が存在しないことを意味します。

重要なのは、Option<T> の値をそのまま T 型として直接使用することはできない、という点です。通常は match 式などを使い、タグが Some の場合にのみ値へアクセスするように記述します。この制約により、不適切な使用はコンパイル時にエラーとなり、安全性が保たれます。

以下はその例です：

```
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
```

ここで v は、2つの Option<Box<i32>> 値からなる配列です。1 つは None、もう 1 つは、ヒープ上の値として 100 を格納した Box<i32> ポインタを含む Some バリアントです。

Option を処理するには、match 文で行うように、その「タグ」を調べます。Option を、その Some バリアントの型の値として直接使おうとすると、コンパイル時エラーになります。

```
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
```

所有権、動的メモリ割り当て、ドロップ処理

前述のとおり、Rust では Box<T> のような、動的メモリ割り当て専用の標準的なポインタ型が用意されています。この型のポインタは、初期化されていない場合を除き、自身が指す値の所有権を持ちます。また、後述する参照カウント型を除き、所有権を持つポインタは一意であり、他のポインタと所有権を共有することはできません。
代入（パラメータ渡しやフィールド初期化のような暗黙的な場合も含む）によって、値の所有権はソースからターゲットへ移動します。Rust の用語では、この操作を「ポインタがソースからターゲットに ムーブ（移動）される」と表現します。唯一の例外は、ソースとターゲットのポインタが同じ場合の自己代入（“no-op” の特殊ケース）です。

単一所有権ポインタのムーブは、通常、以下のステップを含みます:

- ターゲットが有効（すでに初期化済み）のポインタである場合、Rust のランタイムは、ターゲットが参照していたヒープ上の値を自動的に解放（"drop"）します。この処理は、値の中にさらに単一所有権のポインタが含まれている場合には、再帰的に行われます。
- ソースポインタの内容がターゲットにコピーされます。
- ソースは未初期化として扱われ、再初期化される前に参照しようとすると、コンパイル時エラーとして検出されます。

初期化された単一所有権ポインタ変数の宣言を含むスコープの終わりで、Rust ランタイムは、当該ポインタが参照しているヒープ上の値を自動的に 解放（drop）します。この処理は、値の内部にさらにヒープ所有ポインタが含まれている場合には、再帰的に行われます。
ポインタ変数は、それを含むスコープに所有され、そのスコープのスタックフレームが破棄される際に、暗黙的に解放されます。

なお、「ポインタ変数」という用語は、関数の仮引数や構造体フィールド、配列やベクタの要素などに含まれるポインタも含む広い意味で使用されています。

Rust の「単一所有権」方式は、ガベージコレクタを使わずに到達不能なメモリ領域を自動的に回収します。また、ダングリング参照（dangling reference）を防ぎ、ヒープ上の値の破損も避けます。（ヒープ上のオブジェクトが 1 つの所有者しか持てない仕組みになっているため、別のスレッドに属するポインタがそのオブジェクトを勝手に所有することはできません。）
さらに、この仕組みにより **Vec<T>** のような型を効率的に実装できます。Vec<T> は容量を超えた場合に再割り当てが必要になりますが、所有権ルールのおかげで安全に処理できます。

しかしながら、単一所有権モデルだけでは制約が強すぎる面もあります。

- ポインタを関数に渡すたびに所有権を移動させる必要があるため、関数呼び出し後もポインタを使いたい場合には、煩雑な書き方を強いられます。

- 単一所有権の制限は、一部の一般的なデータ構造の実装を妨げるだけでなく、宣言済み変数へのポインタ（動的に割り当てられた値へのポインタではない）を使う間接参照の利用をサポートしていません。

この制約に対応するため、Rust の安全なポインタ機構には以下の要素が用意されています。

間接参照、参照、借用チェッカ（Borrow Checker）

Rust で、おそらく最も革新的であり、同時に一番難しく感じられる機能は、「間接参照に使えるポインタの扱い」です。これを **参照（reference）** と呼びます。構文は C 言語のポインタに似ていますが、新しいルールや工夫が加えられています。

- &x は x の不変借用（(immutable borrow)です。これは、可変または不変の値 x への参照を作成し、x を読み取ることはできますが、変更することはできません。x の所有権には影響しません。 & 演算子は、宣言された変数、ヒープ上の値、リテラル、関数名など、実行時に存在する構造に適用できます。
- &mut x は x の可変借用（mutable borrow）です。これは &x に似ていますが、x を変更することができ、不変の値には適用できません。
- &T は、x が型 T のときの &x の型です。つまり、型 T の値を参照できますが、変更はできないポインタの型です。
- &mut T は、&mut x の値に対応する型で、参照された値を変更できます。

安全性の低いプログラミング言語では、このような機能を使う際に、参照先が無効になる「ダングリング参照」やデータの破損といったリスクが生じるほか、プログラムの最適化を妨げる原因にもなります。Rust では「借用チェッカ（Borrow Checker）」と呼ばれるコンパイラの仕組みによって、この問題を未然に防ぐための制約が自動的に適用されます。

ダングリング参照を防ぐ例：

```
{                          // 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
```

行(1) では ptr が i32 型への不変参照として宣言され、行(2) で変数 n を参照します。この時点では、ptr は潜在的にダングリング参照となる可能性がありますが、行(3) における使用は問題なく許容されます。しかし、行(5) で *ptr を使用しようとすると、行(4) の時点で n はすでに破棄されているため、ptr は実際のダングリング参照となってしまいます。Rust の借用チェッカ（Borrow Checker）は、ライフタイム解析がエラーを検出し、コンパイルを停止します。

行 (5) がなければ、ptr のライフタイムは行 (1) から最後の使用箇所「行 (3)」までに限定されます。ptr 自体はスコープの終了「行 (6)」まで破棄されませんが、ライフタイムの観点で使用が終わった時点で終了と見なされます。
しかし、行 (5) が存在する場合、ptr のライフタイムは行 (4) まで延長されます。これは参照先の n のライフタイムよりも長くなるため、このプログラムは違法となります。
変数のライフタイムがその字義的スコープ（lexical scope）より短くなる可能性があるという性質（「非字義的ライフタイム（non-lexical lifetimes）」と呼ばれる）は、安全性を損なうことなく柔軟性を提供します。

不変な変数は（immutable variable）、不変にしか借用できません。可変な変数（mutable variable） x は、&mut x によって可変に、あるいは &x によって不変に借用することができます。ただし、後者の場合（&x を使った場合）でも x を変更することは可能ですが、それは &x ポインタ経由ではできません。

Rust で、ポインタの安全性の要となるのは、「可変借用の排他性」です。変数が不変に借用されている間は、その他すべての借用（可変・不変を問わず）が禁止されます。一方で、不変な借用であれば、同時に複数存在しても問題ありません。変数を直接使用する場合も、借用と見なされます。
マルチスレッド環境において、この性質は一般に「同時読み取り、排他的書き込み（Concurrent Read, Exclusive Write）」、CREW として知られています。Rust はこの原則を、読み書きロックやミューテックスなどのさまざまな機能を通じて強制するとともに、ダングリング参照の発生も防いでいます。

スレッド間での共有データの直接使用は可能で、静的データは thread::spawn() に渡される関数やクロージャから参照できますが、可変の値については推奨されていません。静的変数の可変借用の排他性を言語レベルでチェックする仕組みは存在しません。

そのため、可変静的変数を使用する場合は、すべて特別な構文(「unsafe」ブロック)の中で記述する必要があります。プログラマが干渉のないことを自ら保証しなければなりません。一方、不変の静的変数は安全に共有できます。
可変静的変数を除き、可変借用の排他性は単一スレッド内で強制されます。これにより、エラーが起こりやすい構造を防ぎつつ、最適化が可能になります。

以下は、Jon Gjengset の著書『Rust for Rustaceans』からの例です:

```
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
         }
}
```

借用チェッカ(Borrow Checker)は、行(1) の呼び出しを許可します。これは m と n が異なる変数であるためです。一方、行(2) のコードは n を不変と可変の両方で借用しようとしているため、拒否されます。
この制約により、コンパイラは関数の冒頭で *input の値を安全にキャッシュすることができます。なぜなら、input と sum が同じ変数を参照することはないからです。この記事の前半で紹介された C の関数と比較すると、最適化によってプログラムの動作が変わってしまう可能性があることがわかります。

ちなみに、ポインタの安全性を保証するための「所有権」や「借用」の概念は、Rust に固有のものではありません。SPARK 言語にも似たような概念があります。ただし、SPARK ではポインタ(「アクセス値」と呼ばれる)に制限があり、プログラムを形式検証することが可能です。

参照カウント型ポインタ(Reference-counting pointers)

単一の所有権(single ownership)はメモリ管理を簡素化しますが、1 つの値に対して 1 つの所有者しか持てないという制約は、場合によっては厳しすぎることがあります。たとえば、有向非巡回グラフ(directed acyclic graph)では、複数のノードが同じノードを指すことがありますが、

明確な所有者が存在しない場合があります。このような参照されるノードは、最後の所有者が破棄されたときにのみメモリを解放すべきです。
このような状況は、古典的な参照カウント（reference count）によって管理できます。Rust では、これを実現するために、スマートポインタ型である Rc<T> と Arc<T> が用意されています（Arc の "A" は "Atomic" の略です）。
これは Box<T> と同様に、型 T の値をヒープに動的に割り当てることができますが、Rc<T> や Arc<T> では、ヒープ上の T の値に所有者の数をカウントする情報が含まれています。
Rc<T> はスレッド安全ではないため、単一スレッドのコードでのみ使用できます。
Arc<T> は必要な保護機能を備えており、パフォーマンスの低下を伴いますが、マルチスレッドコードで安全に使用できます。

参照カウント型ポインタの代入には、Rust の標準的なムーブセマンティクス（“move” semantics）が適用され、所有権は元の変数から新しい変数へ移動します。ただし、通常のポインタと異なり、代入によって参照先のヒープメモリが即座に解放されるのではなく、参照カウントが 1 減少します。参照カウントが 0 になると、メモリが解放されます。
参照カウント型ポインタで値の所有権を共有したい場合は、Rc<T> や Arc<T> の clone() 関数を使用します。名前は「クローン」ですが、ヒープ上の値をコピーするのではなく、ポインタをコピーするだけです。

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

代入文に含まれる 3 つの動作：

- Rc::clone(&p) は単にポインタ p を返し、*p の参照カウントを 1 増やします。
- *q の参照カウントが 1 減り、0 になったため、*q のメモリが解放されます。
- Rc::clone() の結果（ポインタ p）が q にコピーされ、*p は p と q によって共有所有されるようになります。

Rc<T> または Arc<T> ポインタ p が参照している値の参照カウント（正確には強い参照の数）は、それぞれ Rc::strong_count(&p) または Arc::strong_count(&p) 関数によって取得できます。

Rust の原則である「ある値が 1 つのポインタから変更されている間に、他のポインタから読み取りや変更が行われてはならない」というルールを守るため、Rc<T> や Arc<T> が参照する値は不変（immutable）でなければなりません。これは大きな制約ですが、後述するように Rust はこの制約を回避する手段も提供しています。

以下は、参照カウント方式のポインタを用いた複数所有の例です。不正な行にはコメントが付けられています。

```
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 );

}
```

内部可変性（Interior mutability）

参照カウント付きの値に対する不変（immutability）という要件は十分に理に、かなっていますが、制約が厳しすぎる場合があります。そこで Rust は、本来は不変である値の内部に可変性（mutability）を持たせるための明示的な仕組みを提供しています。安全性は保持されており、可変参照の排他性はチェックによって強制されますが、このチェックは実行時に行われます。チェックが失敗すると、影響を受けたスレッドはパニックを起こして終了します。

本来は不変である値の内部に可変性を持たせる機能は「内部可変性（interior mutability）」と呼ばれ、Cell<T> 型と RefCell<T> 型によって実現されています。Cell<T> には get() と set() メソッドがあり、T がムーブではなくコピー可能な値（たとえば i32 のようなスカラー型）の場合に適しています。それ以外の場合は RefCell<T> を使用します。RefCell<T> には borrow() および borrow_mut() メソッドが用意されています。

内部可変性という特性は、より一般的に適用できます。たとえば、構造体型を定義する際に、あるフィールドは不変、別のフィールドは可変とすることが可能です。中でも、この特性は参照カウント付きの値に対して特に有用です。

以下は、実行時に借用違反が検出される例です。

```
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 );
}
```

行(1)では、ptr1 と ptr2 はともに RefCell 型の値 100 を指し、その所有権を共有しています。この値はヒープ上に割り当てられており、参照カウントは 2 です。行(2)では、この値に対して可変借用が行われます。行(3)では、同じ値に対して再び(今度は不変で)借用を試みますが、実行時エラー(パニック)となり、プログラムが終了します。

この問題の修正方法は、2 番目の借用を行う前に、最初の可変借用(borrow_mut)がスコープから抜けて解放されることです。

以下が修正されたコード例です:

```
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 );
}
```

このプログラムは、先ほどのバージョンと同様に始まり、行(1)で RefCell の所有権を共有し、行(2)で値を可変に借用します。
しかし、可変借用（borrow_mut）のスコープは行(3)で終了するため、その後の行での借用は許可されます。
このプログラムは実行時エラーなしで正常に実行されます。

弱い参照（Weak references）

複雑なデータ構造では、個別に割り当てられたノード同士が互いに直接または間接的に参照し合うことで、循環（サイクル）が生じることがあります。
このようなサイクルは参照カウント方式にとって課題となります。というのも、ノード間に相互依存があると、各ノードの参照カウントが 0 にならず、メモリが解放されないためです。
Rust では、循環するデータ構造の定義を可能にするために、参照カウント付きポインタを次の 2 種類に区別しています：

- 「強い（strong）」参照はメモリ解放の判断に影響します。Rc や Arc ポインタのデフォルト動作であり、強い参照のカウントが 0 になった時に、ヒープ上の値は解放されます。

- 「弱い（weak）」参照はメモリ解放には影響しません。弱い参照のカウントが 0 でなくても、強い参照のカウントが 0 であれば、その対象は解放されます。

強い参照は、弱い参照に「ダウングレード」できます。逆に弱い参照は、有効な強い参照が存在していれば「アップグレード」して強い参照に戻すことができます。
弱い参照（weak reference）は、データ構造のノード間に自然な「親子」関係がある場合に便利です。親は子への強い参照（strong reference）を持ち、子は親への弱い参照を持ちます。
一例として、各ノードがデータフィールド（例えば i32）と 2 つのポインタ（実際には Option 値）を持つ双方向リンクリストが挙げられます。

- next ポインタ：末尾ノードでは None、それ以外では後続ノードを参照する Some バリアント。

- prev ポインタ：先頭ノードでは None、それ以外では前のノードを参照する Some バリアント。

prev リンクと next リンクを組み合わせることで循環参照が発生する可能性はありますが、next リンクを強い参照（strong reference）、prev リンクを弱い参照（weak reference）として定義することで、適切にメモリを解放できます。データ構造に用意されたメソッドの実装により、next リンク自体が強い参照の循環を作らないことが保証されています。

以下に、関連するデータ構造の Rust 定義と、メソッドのサンプルの実装を示します：

```
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);
}
```

双方向リンクリストにおいて重要な(暗黙の)性質は、強い参照の循環が存在しないことです。もしアプリケーションコードが、あるノードの next 参照を自分自身や前のノードに向けてしまうような双方向リンクリストを作ると、強い参照カウントがゼロになることはありません。

ストレージ(メモリ)リークを防ぐには、次の 2 つの方法があります:

- 循環を構成するノードの next リンクのいずれかを None に設定して、明示的に循環を断ち切ると、自動的にメモリが解放されます。

- データ構造を生ポインタ(raw pointer、後述)で定義し、手動でメモリを解放します。

生ポインタ（Raw Pointers）

低レベルのプログラミングや、安全なポインタや独自のメモリ管理の実装の基盤として、Rust は 生ポインタ（raw pointers） と呼ばれる仕組みを提供しています。
これは C 言語のようなポインタ機能を提供しますが、同時に C 言語と同様の安全性チェックの欠如も伴います。

Rust の生ポインタには以下の 2 種類があります：

- *const T（不変ポインタ）：参照先の T の値を読み取ることは可能ですが、書き換えはできません。

- *mut T（可変ポインタ）：参照先の T の値を読み書き可能です。

生ポインタ（raw pointers） は、通常の参照（& 演算子を使う）や動的メモリ割り当てによって作成できます。また、生ポインタと安全なポインタの間では相互に型変換（キャスト）が可能です。ただし、生ポインタを実際に参照外し（dereference）できるのは unsafe ブロック の中だけです。さらに、もし標準ライブラリの std::alloc モジュールにある alloc() 関数を使って生ポインタ用のメモリを割り当てた場合、そのメモリはプログラム側で dealloc() 関数を呼び出して明示的に解放する必要があります。

以下に例を示します：

```
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);
    }
```

生ポインタ（raw pointer）は効率的であり、必ずしも危険（unsafe）とは限りません。例（コメント付きの行 (1) と行 (2) ）に示すように、Rust では一部のコンパイル時チェックによって危険な使い方を防いでくれます。
しかし、安全なポインタが持つ保証がないため、開発者はコードの正しさを自分でより厳密に確認する必要があります。生ポインタでは、従来から知られているすべてのエラー（ダングリング参照、メモリリーク、二重解放、ヌル参照など）が発生し得ます。

生ポインタの重要な用途のひとつは、メモリプールやカスタムの割り当て・解放機構を定義するためのツールキットとして使うことです。これは特に組込系アプリケーションで有用です。
Rust コミュニティの crates.io レジストリには、typed-arena、slab、mempool などのクレートが存在します。

生ポインタは、言語による安全性チェックを回避するための安易な手段として（誤って）使うべきではありません。むしろ、外部（非 Rust）コードとのインタフェースや、低レベル機能の実装といった、正当な用途のために使うべきです。

モグラ退治の再訪（Revisiting the mole）

Rust はポインタというモグラを完全に退治できたのでしょうか？
Rust プログラマは、安全で効率的なコードを書きながら、基盤となるデータ構造を安心して定義できるのでしょうか？
その答えはやや限定付きではありますが「はい」です。

良い点として、Rust は多くのポインタ関連のエラーをコンパイル時に検出し、汎用的なガベージコレクタを使うことなく自動的にメモリを回収します。また、型定義機能も汎用的に活用できる柔軟さを備えています。
ただし、ポインタ機構に本質的に伴う緊張関係やトレードオフを考えると予想できるように、Rust のアプローチにもいくつかの欠点があります。

Rust を学ぶ上で最も大きなハードルのひとつは、ポインタの仕組みを理解するための学習曲線だと言えるでしょう。借用チェッカ（Borrow Checker）は、ソースコードの流れを解析して動くアルゴリズムです。その考え方は、「値は可変なら排他的に、不変なら共有して借用できる」というシンプルなルールにまとめられます。ただし、実際に使う場面では思わぬ制約に出会うこともあり、二重リンクリストの例に見られるように、従来のデータ構造の書き方ではなく、少し工夫したスタイルが必要になることもあります。

ここで、方法論的には同等の 2 つの例を紹介します。どちらも、借用した参照を通じて変数を変更するコードです。しかし、一方は正当で、もう一方は不正です。まずは不正なコードから見ていきましょう。

```
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)
}
```

行(1) で、p は n を可変に借用しています。p のライフタイム（およびそれに伴う n の可変借用の有効範囲）は 行(3) まで続きます。
しかし、行(2) の println!() マクロでの n の使用は、暗黙の不変借用にあたります。
この不変借用が、可変借用の有効範囲内で発生しているため、このコードは不正となります。

```
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)
}
```

違いは、変数 p のライフタイムが行 (2) で終わっていることです。そのため、行(3) で行われている n の不変参照 は、すでに p の可変参照の有効期間 とは重なっていません。
この 2 つのコード例の違いは、一見すると分かりにくいかもしれません。
変数を 間接的に変更（可変参照経由）しながら 直接参照（不変参照） するようなケースでは、それが正当か不正かは状況によって変わります。
さらに、内部可変性（interior mutability） が必要な場合には、借用チェックのルールが 実行時に適用されることになります。その際、借用違反によるパニックを避けるために複雑な解析 が必要になる場合もあります。

もう一つの潜在的な問題は、複雑なデータ構造が破棄されるときのメモリ解放コスト です。
C、C++、Ada などの明示的なメモリ解放を行う言語では、危険が伴うとはいえ、解放処理はソースコード上で明示されており、特に「固定サイズのブロックを持つメモリプール」を使う場合には、その実行時コストをある程度予測できます。
一方 Rust では、メモリ解放は 暗黙的に行われる ため、その実行時コストを見積もるには慎重な解析が必要です。

Rust には「カスタムストレージプール機構」が用意されており、この問題を軽減できますが、その代わりに 手動解放 を伴い、従来のリスクも再び生じることになります。

これらの問題点はあるにせよ、Rust は表現力の高いポインタ機能を備えており、安全で効率的なコードを書くための技術と実践を大きく前進させています。
Rust は他の言語に比べて習得に時間と労力がかかります。しかし、コンパイル時に多くのポインタ関連エラーを検出することで、メモリ安全性を確保 し、ホーア（Hoare） 教授が指摘した「10 億ドルの失敗（null ポインタ問題）」を回避し、さらに検証コストを削減できます。
ポインタという厄介な“モグラ叩き”問題は完全に解決されたわけではありませんが、Rust はそのモグラをほぼ叩き伏せ、無力化することに成功しています

著者に関して

ベンジャミン・ブロスゴル博士（Dr. Benjamin Brosgol）は、AdaCore 社のシニアテクニカルスタッフの一員です。これまでのキャリアを通じて、高信頼性ソフトウェア向けのプログラミング言語技術に注力してきました。
Ada 95 の設計チームの一員であり、リアルタイム Java 仕様（Real-Time Specification for Java）を開発した専門家グループのメンバーでもあります。
ブロスゴル博士は、安全性・セキュリティ規格（DO-178C、Common Criteria）やプログラミング言語（Ada、Java、C#、Python）に関する論文を発表し、チュートリアルも行っています。また、AdaCore を代表して FACE®コンソーシアム（Future Airborne Capability EnvironmentTM）に参加し、その技術作業部会の副議長も務めました。
ブロスゴル博士は、アマースト大学で数学の学士号を取得し、ハーバード大学で応用数学の修士号および博士号を取得しています。

※本資料は、AdaCore のテクニカルペーパーを意訳したものです。
正確な内容については、原文をご参照下さい。
https://www.adacore.com/papers/mission-critical-rust-managing-memory

IT Access®
アイティアクセス株式会社

本社　〒222-0033　神奈川県横浜市港北区新横浜 3-17-6
TEL:045-474-9095 FAX:045-474-8823
URL: https://www.itaccess.co.jp/service/adv/brand/adacore/

202508