アンセーフ

アンセーフコード

本来 C# 言語の型は値型、または参照型のいずれかであり
参照型はガーベジコレクタによって、自動的にメモリ管理されます

ところが、C# には**ポインタ型**が存在しています
C/C++ と同様に、C# 言語もまた、低水準で高度なメモリ管理を提供しています
このような、ポインタを用いた C# コードを**アンセーフコード**と呼びます

アンセーフコードは、ガーベジコレクションの対象外となります
すなわち、ポインタの扱いは従来どおり、プログラマの手腕に委ねられるのです
通常のプログラムでは、アンセーフコードを用いるメリットはありません
それならば、C++ 言語を用いて .NET プログラムを行ったほうが効率的でしょう

しかし、アンセーフコードは直接ポインタを扱うわけであり
ガーベジコレクタの管理対象メモリ領域に配置される変数よりも高速に動作します

アンセーフコードは、特定の**アンセーフコンテキスト**でのみ使用できます
アンセーフコンテキストとは **unsafe** 修飾子が指定されているブロックを指します
このキーワードは、クラスや構造体、インターフェイス、及びメンバに修飾できます
同時に unsafe キーワードを使用するプログラムは、コンパイルする時に
/unsafe オプションを指定しなければなりません

C# 言語の型は、値型と参照型のみでしたが、
アンセーフコンテキストでは、これにポインタ型が追加されます
ポインタ型は、**アンマネージド型**か**void**キーワードと
それに続く ***** 記号のトークンで構成されます
この仕様は、C/C++ 言語とまったく同じなので、従来の言語に慣れていれば親しみやすいでしょう

ポインタ型において * の前に指定される型のことを、ポインタ型の**リファレント型**と呼びます
リファレント型は、常にポインタが指す変数の型を指定します
すなわち int 型のポインタ型は int * であり、リファレント型は int です

しつこいようですが、ポインタはガーベージコレクションの対象外です
すなわち**ポインタは参照を指せない**ということを理解してください
参照型はガーベージコレクションの追跡対象です
とうぜん、参照型を含む構造体なども指すことはできません

.NET における変数は**固定変数**と**移動可能変数**があり
固定変数とは、ガーベージコレクタの対象外の全ての変数のことを表します
すなわち、ローカル変数や値パラメータなどです
固定変数の特徴は**変数のメモリアドレスが不動**であることです

これに対し、移動可能変数とはガーベージコレクション対象記憶領域に位置する変数で
オブジェクトのフィールドや配列の要素などはこれに属します
移動可能変数は、ガーベージコレクタが自動的にアドレスを変更します
メモリは、確保と解放を繰り返すことによってフラグメンテーションが発生するためです

ポインタ型が受け取るアドレスは、常に固定変数のアドレスでなければなりません
もし移動可能変数のアドレスを受けたとしても、逆参照する時に存在が保証されません
そのため、C# では、このようにポインタに対する規制も厳しく設定されているのです

固定変数のアドレスを得るには、アンパサンド記号 **&** を用います
これを**アドレス式**と呼び、アドレス式は常に & と単項式で構成されます
アドレス式は、指定した単項式のメモリアドレスを返します
この値を、互換性のあるポインタ型変数で受け取ることができます

ポインタ変数が指すアドレスの値を参照するには**間接参照式**を用います
間接参照式は、アスタリスク * にポインタ型の単項式を指定します
間接参照式は、ポインタ型のリファレント型の値を返します

```
class Test {
 public static unsafe void Main() {
  int x = 10;
  int * po = &x;
  System.Console.WriteLine(&x);
 }
}
```

このプログラムの Main() メソッドは unsafe 修飾子が指定されています
これは、アンセーフコードなので、コンパイル時に /unsafe オプションを指定してください

ポインタ型変数 po は、int 型の固定変数のポインタを取得できます
変数 x のアドレスを & 記号を用いたアドレス式で取得していることがわかりますね

ポインタを通じて、構造体のメンバにアクセスするには -> 演算子を使います
これについても、C/C++ に慣れている人にとってはお馴染みでしょう

```
struct Point {
 public int x , y;
```

```
 public Point(int x , int y) {
  this.x = x;
  this.y = y;
 }
}

unsafe class Test {
 public static void Main() {
  Point pt = new Point(400 , 300);
  WritePoint(&pt);
 }
 public static void WritePoint(Point *pt) {
  System.Console.WriteLine(
   "x = " + pt->x + " : y = " + pt->y
  );
 }
}
```

このプログラムは、座標を指定する構造体 Point を定義しています
Main メソッドでは Point 構造体を宣言、初期化しています
WritePoint() メソッドは、Point 構造体のポインタを受け取り、値を出力します

移動可能変数のポインタ

通常、移動可能変数のポインタを取得することはできません
移動可能変数は、そのアドレス自体がガーベジコレクタによって移動させられるため
ポインタそのものに意味が無くなってしまうのです

そこで、移動可能変数のポインタを取得するには
移動可能変数のアドレスを、一時的に固定させる必要があります
これを行うには **fixed** ステートメントを用います

fixed (type* ptr = expr) statement

type にはリファレント型を、ptr にはポインタ変数名を指定します
expr は、ポインタ変数に代入する移動可能変数のアドレス式を指定します
後は、statement に必要な処理コードを指定します

移動可能変数は fixed のステートメントを実行している間は、アドレスが固定されます
これによって、一時的に移動可能変数のアドレスが保証され、ポインタが使えます
fixed ステートメントブロックの間、このポインタは有効です

例え静的変数でも、フィールドは移動可能変数に属します
そこで、フィールド変数のポインタを得るには fixed を使います

```
unsafe class Test {
 public static int param = 10;
 public static void Main() {
  fixed(int *pParam = &param)
   System.Console.WriteLine(*pParam);
 }
}
```

このプログラムは、Text クラスの param フィールドのポインタを取得し
fixed ステートメントを用いて、一時的にメモリの再配置を防いでいます

---

unsafe

アンセーフコンテキストであることを表す修飾子です
クラス、構造体、インターフェイス、及びメンバに指定することができます

fixed (type* ptr = expr) statement

値型、またはポインタ型の移動可能変数のポインタを取得し
CLR にアドレスを変更させてはいけないということを知らせます

type - ポインタのリファレント型を指定します
ptr - ポインタ変数名を指定します
expr - 移動可能変数のアドレス式を指定します
statement - 埋め込みステートメントを指定します
