ポインタ

ポインタの型キャスト

アンセーフコンテキストでは、ポインタ型のキャストも可能になります
ポインタ型のキャストにも、暗黙的に行えるものと、そうでないものがあります

ポインタ型は、明示的にキャストすることで、非常にユニークな使い方ができます
C/C++ 言語の経験者にとっては、お馴染みの作業かもしれませんが
Java から C# 言語に移行してきた人であれば、ポインタの威力を肌で感じることでしょう

ポインタ型は、明示的にキャストすることで、**他のポインタ型に変換**できます
この変換作業によっては、メモリの直接アクセスにでしか行えない面白いことができます
例えば、次の例は数値型変数のポインタを char * 型として扱っています

```
unsafe class Test {
 public static void Main() {
  int i = 0x41;
  char *c = (char *)&i;
  System.Console.WriteLine(*c);
 }
}
```

int 型変数 i のポインタを char 型のポインタ変数に変換していることが確認できます
WriteLine() メソッドに、char 型のポインタ c の関節参照の値を渡すとどうなるのでしょう
ポインタが指す実体は int 型の変数で 0x41 という値を持っています

しかし、char 型のポインタに変換したポインタ変数の間接参照は
char 型のメモリサイズを対象とし、0x41 という文字コードを持つ char 型として扱われます
ASCII 文字コードで 'A' をあらわすため、このプログラムは A を出力します

この機能を用いれば、世にも恐ろしいソースを記述することができます
ポインタのキャストは、メモリアドレスの実体に関係無く
そのアドレスが指すメモリの情報を指定した型に無理やり当てはめてしまうため
次のような強硬手段を用いることも可能となっています

```
struct Location {
 public int x , y;
}

struct Size {
 int width , height;
 public void Write() {
  System.Console.WriteLine(width + "¥n" + height);
 }
}

unsafe class Test {
 public static void Main() {
  Location lt = new Location();
  lt.x = 100; lt.y = 200;

  Size *sz = (Size *)&lt;
  sz->Write();
 }
}
```

警告が出るかもしれませんが、問題なくコンパイルすることができます
このプログラムは Location 構造体の lt 変数を Main() メソッドで作成し、メンバを設定します
そして、このプログラムは恐ろしい強硬手段を用います

Location 構造体と Size 構造体のフィールドの物理サイズは同じです
この性質を利用して、Location 構造体の変数を Size 型のポインタにキャストして代入します
sz ポインタ型変数が指す先は、Location 構造体のメモリアドレスです
しかし、キャストによって、Size 構造体を用いているかのように錯覚させることができます
もちろん、特別な緊急事態を除いてこのようなプログラムを書いてはいけません

さらに、ポインタ型は**整数型にキャスト**することが可能であり
整数型は**ポインタ型にキャスト**するということが可能になっています
すなわち、メモリアドレスの値を直接見たり、数値をアドレスとして使うことができるのです

```
unsafe class Test {
 public static void Main() {
  int x = 0x10;
  System.Console.WriteLine((int)&x);
 }
}
```

これを実行すれば、標準出力には何らかの値が表示されるでしょう
ここで表示される値こそ、ポインタ変数が格納しているアドレスそのものです
もちろん、この数値が表す意味自体は、プログラマが知る必要はありません

このアドレス値は、キャストすることで通常の整数型変数に代入することも可能であり
その整数型変数をキャストして、ポインタ型変数に代入することも可能です

高度なオブジェクト指向プログラムでは、普段、そのようなことをしてはいけません

なぜならば、数値型変数が保有する値が本当に有効なメモリアドレスかどうか
これを確かめる術はなく、場合によっては不正なメモリにアクセスしかねないからです

しかし、C# 言語はネイティブなシステムの API を呼び出す機能があり
システムの API と互換性を取るために、ポインタによる演算を行うことがあります
以下のような斬新なプログラムも、C 言語ではよくある演算です

```
using cout = System.Console;

struct Msg {
 public const int INT = 0x2;
 public const int KITTY = 0x4;

 public unsafe static bool Proc(uint msg , int lParam) {
  switch(msg) {
  case INT:
   cout.WriteLine(*((int *)lParam));
   return true;
  case KITTY:
   ((Kitty *)lParam)->Write();
   return true;
  default:
   cout.WriteLine(lParam);
   return false;
  }
 }
}

struct Kitty {
 public void Write() {
  cout.WriteLine("Kitty on your lap");
 }
}

unsafe class Test {
 public static void Main() {
  int x = 0x10;
  Kitty cat = new Kitty();

  Msg.Proc(Msg.INT , (int)&x);
  Msg.Proc(Msg.KITTY , (int)&cat);
  Msg.Proc(0 , 0x100);
 }
}
```

このプログラムの Msg.Proc() メソッドは Win32 のウィンドウプロシージャを思い出させます
msg には、意味のあるシステムからの通知を受ける値だと仮定し
lParam は、通知ごとに意味の異なる、追加情報だと想定してください

システム設計理論から考えて、通知の追加情報は動的であるべきです
システムが通知するメッセージによっては、膨大な情報量が必要な時もあるでしょう

lParam 追加情報の意味は msg によって一意に定まります
システムはあらかじめ、メッセージに関連した lParam の情報を定めます
そして、システムは構造体をメソッドに渡したい時、ポインタをキャストして渡すのです
これは、実際に Windows API が取った手段です

メッセージを受けるメソッドは、msg パラメータを調べれば
システムが何を要求しているかを知ることができます
そして、メッセージによって lParam に何が格納されているかを知ることができるのです
必要によっては lParam に格納されている値をキャストして、間接参照を行います

上の例では、Main() メソッドから INT と KITTY メッセージを通知します
INT メッセージの場合、lParam には int 型のポインタが格納され
KITTY の場合は Kitty 構造体のポインタが格納されていると定義します

Proc メソッドは、switch() によってメッセージを解析し
それぞれのメッセージの専用処理を用意し、これに対応しています
このプログラムを実行すれば、次のような文字が出力されるでしょう

16
Kitty on your lap
256

この結果を見れば、アドレスが正しくキャストされていることがわかります

汎用ポインタ

C# 言語における参照型の Object クラスのように
あらゆるポインタ型を代入することができるポインタ型が存在すれば便利です

この要求は **void *** 型が解決してくれるでしょう
これは、C 言語で言う「汎用ポインタ」とまったく同じで、全てのポインタを代入できます
ポインタ型の void * 型へのキャストは、暗黙的に行えます

先ほどの Msg.Proc() メソッドは、数値型のパラメータがあり
必要に応じて、数値を渡したり、ポインタのアドレスを渡す方法が用いられましたが
型がわからないが、ポインタ型であるということが確定している場合は void * 型を使います

このポインタ型は未知の型へのポインタとして用いることができるのです
当然ですが、間接参照を行うには明示的に型キャストしなければ使えません

```
unsafe class Test {
 public static void Main() {
  char c = 'C';
  int i = 0x10;

  WritePointer(&c);
  WritePointer(&i);
 }
 public static void WritePointer(void *po) {
  System.Console.WriteLine(*((uint *)po));
 }
}
```

このプログラムの WritePointer() メソッドは、あらゆるポインタ型を受け取れます
受け取ったポインタは、uint 型のポインタと解釈し、整数で内容を出力します

前のページへ　戻る　次のページへ