スタック

ポインタの演算

アンセーフコードにおいて、配列のポインタを格納することはできませんでした
C# の配列は、C のような物理的に連続した配列ではないからです

しかし、アンセーフコードは物理的に連続したメモリ領域をサポートしています
このような領域を得るには **stackalloc** キーワードを用います
このキーワードは、連続したスタック領域のポインタを返してくれます

type * ptr = stackalloc type [expr];

type にはポインタのリファレント型を、ptr はポインタ変数の名前を指定します
expr は、スタックに割り当てる連続したメモリの配列数を指定します
スタックは明示的に解放することはできず、メソッドの処理を抜けた時に解放されます

これで、ptr は連続した物理メモリの配列の、先頭へのポインタということになります
C# のポインタも、C 言語同様に**ポインタの演算**が可能であり
stackalloc で割り当てたメモリへのポインタは、演算によって適当な位置にアクセスできます

ポインタの演算には、++、--、+、- の算術演算子と比較演算子を用いることができ
ポインタをインクリメントするということは、配列の要素を1進めることを意味します
これは、C 言語同様に、ポインタのアドレスに 1 加算すると言う事が
単純にアドレスの値を 1 増やすことではないことを表します

4 バイト型のポインタをインクリメントすれば、それはアドレスに 4 を加算するということです
ポインタ型 p の要素 p[3] にアクセスしたい時 ***(p + 3)** を指定します
括弧で括るのは、* 演算子が + 演算子よりも優先順位が高いためです

```
unsafe class Test {
 public static void Main() {
  int * iIndex = stackalloc int [2];
  *iIndex = 0x10;
  *(iIndex + 1) = 0x100;

  System.Console.WriteLine(*iIndex + "¥n" + *(iIndex + 1));
 }
}
```

このプログラムを実行すれば、16 と 256 という数値が表示されます
プログラムは、最初にスタックに int 型2つ分の領域を確保しています
そして、その先頭に 0x10 を、その次に 0x100 という値を保存しています

32 ビットコンピュータの int 型は4バイトです
もし iIndex が 10000 というアドレスを指していれば iIndex + 1 は 10004 を指します

スタックに確保しているメモリ領域は int 型2つ分、すなわち int[2] の配列です
iIndex はこの配列の先頭のアドレスであり、1 を加算すれば iIndex[1] に等しくなります
*(iIndex + 1) というポインタの演算は、2番目の要素のアドレスを表すことになります

これを用いれば、例えば文字列の表現方法を C 言語と互換にできます
C 言語の文字列とは char 型の配列であり、終端を 0 で表すというものでした

```
unsafe class Test {
 public static void Main() {
  char* str = stackalloc char[256];
  SetStringToChar(str , "Kitty on your lap");
  WriteChar(str);
 }
 static void SetStringToChar(char *pstr , string strSet) {
  int iCount = 0;
  for ( ; iCount < strSet.Length ; iCount++)
   *(pstr + iCount) = strSet[iCount];
  *(pstr + iCount) = (char)0;
 }
 static void WriteChar(char *pstr) {
  for(int iCount = 0 ; *(pstr + iCount) != 0 ; iCount++)
   System.Console.Write(*(pstr + iCount));
 }
}
```

このプログラムの SetStringToChar() メソッドは
char 型のポインタを受け取り、そこに指定した string 型の文字列を移植します
string 型は、単一の文字のアクセスにインデクサをサポートしているので、それを用います

string の文字列の変換が終われば、最後に 0 を代入します
これで、C 言語風 char 型文字配列のできあがりです
WriteChar() メソッドは、受け取った char 型のポインタを出力します
この時、文字列の終端は 0、すなわち NULL 文字が見つかるまで表示します

このように、stackalloc キーワードを使えば
C 言語の様に物理的に連続したメモリ領域を直接扱えます
ただし、stackalloc は動的にメモリを取得するものではありません
スタックに確保するメモリサイズは、常にコンパイル時に決定する静的なものです

ポインタの要素アクセス

配列の先頭へのポインタは、上のようにポインタの演算を用いることで
配列上の任意の要素に直接アクセスすることができました

しかし、ポインタは単純に要素アクセスで参照することができます
ポインタの要素アクセスもまた、基本式に [ ] で添え字を指定する形を取ります
通常、この方がソースは読みやすいので、こちらを使うべきかもしれません
例えば、次のプログラムは上のプログラムとまったく同じものです

```
unsafe class Test {
 public static void Main() {
  char* str = stackalloc char[256];
  SetStringToChar(str , "Kitty on your lap");
  WriteChar(str);
 }
 static void SetStringToChar(char *pstr , string strSet) {
  int iCount = 0;
  for ( ; iCount < strSet.Length ; iCount++)
   pstr[iCount] = strSet[iCount];
  pstr[iCount] = (char)0;
 }
 static void WriteChar(char *pstr) {
  for(int iCount = 0 ; pstr[iCount] != 0 ; iCount++)
   System.Console.Write(pstr[iCount]);
 }
}
```

pstr[iCount] という形で、ポインタ変数から直接、要素にアクセスしています
pstr[iCount] というポインタ要素アクセス式は *(pstr + iCount) に等しいと考えられます

---

type * ptr = stackalloc type [expr];

スタックに、指定した型の配列を割り当て
割り当てた配列のポインタを返します

type - 型を指定します
ptr - ポインタ変数の名前を指定します
expr - 割り当てる配列の個数を指定します

前のページへ　戻る　次のページへ