C 言語講習会　第八回

今回は、前回勉強したポインタの続きと、C 言語に限らずあらゆるプログラミング言語で重要になってくる、**配列**を学んでいきます.

1, 配列

配列は同じ型の同じ変数名で、複数の記憶領域を確保するものです
これらの変数リストの数はプログラマが自由に指定することができ、それぞれの配列の値を**要素**といいます

ここではもっともシンプルな配列、**1 次元配列**を作成しましょう

型 変数名［サイズ］

サイズ意外は、普段の変数の宣言と何ら変わりはありません
サイズとは、配列の個数です
たとえば **int ary[2]** とすると、int 型の ary 配列変数が作成され、その配列は ary[0]と ary[1]という領域を持ちます
ary[1]の 1 番などの要素番号の指定を**添え字(インデックス)**といいます

C 言語の配列は 0 番から数えます
宣言のときに指定した数は個数なのです
int ary[2]で実際に作成される配列は **0 番と 1 番の2つ**になります
添え字には必ず括弧［］で囲みます

宣言された配列には、各要素に通常の変数同様に代入や参照を行なえます
アクセスには添え字を用いて要素を指定します

```
#include <stdio.h>

int main()
{
        int ary[3];

        ary[0] = 10;
        ary[1] = 100;
```

```
	ary[2] = 1000;

	printf("%d¥n%d¥n%d", ary[0] , ary[1] , ary[2]);

	return 0;
}
```

配列 ary を宣言し、各要素に数を代入してそれを出力するプログラムです
それぞれの要素は以下のようになっています

	0	1	2
ary	10	100	1000

このように配列を組めば、同種のデータを一連のリストとして扱うことができます
もちろん算術演算も通常どおり可能です

```
#include <stdio.h>

int main()
{
	int ary[3];

	ary[0] = 10;
	ary[1] = 100;
	ary[2] = 1000;

	printf("%d" , ary[0] + ary[1] + ary[2]);

	return 0;
}
```

また、添え字は定数だけでなく**変数で指定することも可能**です
これによって、for ステートメントなどで制御したりユーザーに指定させたりすることができます

```
#include <stdio.h>

int main()
{
	int ary[3];
```

```
        int count;

        for(count = 0 ; count <= 2 ; count++) {
                ary[count] = 10 * (count + 1);
                printf("%d¥n",ary[count]);
        }

        return 0;
}
```

配列要素の内容を上書きすることも通常の変数同様に可能です
配列の値が固定されるようなことはありません

【演習1】

要素数が100個の一次元配列を定義し、前から順に 1,2,3,…と代入して、最後にそれを全て表示させるプログラムを作成しなさい。

（実行例）

Ary = 1 2 3 4 5 6 7 8 ・・・ 99 100

（ヒント）

printf 関数では、要素をひとつずつなら表示することはできますが、すべての要素を一斉に表示させることはできません（要素が少なければ一つの関数でもまとめて表示させることは可能ですが・・・）。つまり、printf 関数を要素数と同じだけ実行しなければなりません。

2, 多次元配列

多次元配列は **配列の配列** のようなものです
まずは 2 次元配列を中心に考えてみましょう
2 次元以上の配列の宣言は、次のようにサイズの指定をもうひとつ増やします

型 変数名 [サイズ 1][サイズ 2]

これで 2 次元配列を宣言することができます
サイズの指定や添え字については、1次元配列と何ら変わりはありません

```
#include <stdio.h>

int main()
{
        int ary[2][2];

        ary[0][0] = 10;
        ary[0][1] = 100;
        ary[1][0] = 1000;
        ary[1][1] = 10000;

        printf("%d¥n%d¥n" , ary[0][0] , ary[0][1]);
        printf("%d¥n%d" , ary[1][0] , ary[1][1]);

        return 0;
}
```

多次元配列の場合は［サイズ］×［サイズ］の領域が確保されます
上のプログラムの配列構造は次のようになっています

ary	0	1
0	10	100
1	1000	10000

通常は 2 次元配列程度までしか使われませんが、3 次元以上の配列を作成することも可能です
しかし、4 次元以上の配列になると膨大なメモリ領域を確保する可能性があるので
良識的な範囲でメモリを使うように、プログラマが注意する必要があります

【演習2】

２つの行列 A=$\begin{pmatrix}3 & 2\\ 4 & 1\end{pmatrix}$ 、B=$\begin{pmatrix}1 & 2\\ 5 & 7\end{pmatrix}$の積を求めるプログラムを作成しなさい

（ヒント）

各行の表示は、

printf(“%d %d¥n”, １列目の要素, ２列目の要素);

のようにすれば表すことができる

3, ポインタ演算

ポインタも変数の一種であるということは話しました
ポインタも他の変数と同様にデータ型などが存在し、変数同様に扱えます

しかし、注意しなければいけない点がいくつかあります
ポインタも変数のように式を使った算術が可能ですが**整数以外はできない**ということを覚えてください
ポインタに浮動小数点などの演算はできませんし、加えることもできません

つまり、ポインタの演算には**整数による加減算のみ**ということです
それ以外の算術は実行できません

それと、ポインタ変数と通常の変数の算術の大きな違いなのですが
インクリメントとデクリメント演算子を使った算術では、ポインタ変数は不思議な結果を出します
次のプログラムを見てみましょう

```
#include <stdio.h>

int main()
{
        int ary[2] = { 1000 , 2000 };
        int *po;

        po = &ary[0];
        printf("po¥t = %x¥n" , po);
        po++;
        printf("po++¥t = %x¥n" , po);
        printf("*po++¥t = %d" , *po);

        return 0;
}
```

環境で出力される結果はばらばらですが
たとえば、このとき po に代入されたメモリアドレスが 10fd00 だとしましょう
すると次のような結果になります

po　　= 10fd00
po++　　= 10fd04

***po++　　= 2000**

普通インクリメントされたことを考えると 1 加算されて 10fd01 になるように思えます
しかし、ポインタのデータ型が int なので整数は 4 バイト（環境で異なる）の長さとして処理されます
つまり配列の要素が 4 バイトごとに並んでいるので、1 加算するとポインタの値も 4 バイト増えるのです

これはデクリメントでも同じことが言え、当然 float や double ならば、そのバイト長に合わせてポインタの値が変化します
※char 型の場合は 1 バイトなので、この場合は通常の値のように加減算されます

もう一つ注意ですが、間接参照でインクリメントする場合には注意が必要です

***po++**

とやると、多くの人の期待に反した結果が得られると思われます
このときは、アドレスが指す変数ではなく**ポインタの値がインクリメントされてアドレスを参照**されます
結果的に *po++ だけではポインタが格納している値をインクリメントしたが、参照する意味がないので処理系によっては警告が出ます
（代入文の場合、=よりインクリメントの方が優先度が低いので結局意味をもちません）

もし、間接参照でアドレスの変数の内容に対してインクリメントやデクリメントしたい場合は以下のようにします

(*po)++;

これでポインタ変数 po が指す値がインクリメントされます

```
#include <stdio.h>

int main()
{
        int ary[2] = { 1000 , 2000 };
        int *po;

        po = &ary[0];
        (*po)++;
```

```
        printf("%d", ary[0]);

        return 0;
}
```

4, 配列とポインタ

より深く、配列とポインタの関係を見てみましょう
C 言語では、配列とポインタの関係は重要とされます

配列変数は、添え字を省略して配列変数名だけにすると、それは**配列の先頭アドレスへのポインタ**を表します
前回少し触れましたが、以前 scanf() 関数に配列名だけを渡していた理由がここにあります
scanfは変数の前にアドレス演算子 & をつけなければいけないお約束がありましたが、配列の場合はつけませんでした
しかしそれは、配列の場合は添え字省略の変数名は配列の先頭アドレスへのポインタだからだったのです

```
#include <stdio.h>

int main() {
        char str[] = "kitty on your lap";

        printf("str[0]の内容¥t¥t= %c¥n" , *str);
        printf("str[0]のアドレス¥t= %x" , str);

        return 0;
}
```

勘のよい人はお気づきかもしれませんが、ということは配列のある要素へ間接参照することも可能です

ということは、ポインタに格納されているアドレスを演算することによって
ポインタから**配列の特定要素にアクセス**できるはずです
ポインタからアドレスを演算して配列に間接参照を試みてみましょう

```
#include <stdio.h>
```

```
int main() {
        int ary[3] = { 10,20,30 };
        int *ary_p;
        ary_p = ary;

        printf("間接参照¥t=%d,%d,%d¥n", *ary_p , *(ary_p + 1) , *(ary_p + 2));
        printf("添え字指定¥t=%d,%d,%d" , ary[0] , ary[1] , ary[2]);
        return 0;
}
```

結果は次のようになります

間接参照 =10,20,30
添え字指定 =10,20,30

こうすれば、配列で添え字指定するのと同じ効果がえられます
勘違いしないでほしいのは、**ary_p に格納されているのは、あくまで ary[0]のアドレス**だけですよ
ary[1]などを指定するときに、*(ary_p + n) のようにしているのは、演算子の優先順位のためです
(*の方が+より優先される)

【演習3】
キーボードから文字列を入力し、それを表示するプログラムを作りなさい. また、余裕のある人は、その入力した文字数を表示させなさい.

(発展)多次元配列に間接参照する

では、多次元配列になるとアドレスの関係はどうなるのでしょうか

実は、多次元配列への間接参照になった瞬間、異様にややこしくなります
まず、生成される型が多次元なので、手作業で演算してアドレスを指定するには多次元では困り

ます
そのため、ポインタにアドレスを代入するときに型キャストする必要があります

pointa = (type *)ary[0];

次に、多次元への添え字指定の計算方法です
***(pointa + (指定一次元添字 * 二次元要素数) + 指定二次元添字)**です
つまり、多次元配列 ary[10][5] のうち **ary[5][3] にアクセスしたい場合**は次のようになります

***(pointa + (5 * 5) + 3);**

実例を見てください

```
#include <stdio.h>

int main() {
        int ary[2][3] = {
                { 10 , 20 , 30 } ,
                { 40 , 50 , 60 } ,
        };
        int *ary_p;
        ary_p = (int *)ary;

        printf("間接参照 = %d,%d,%d¥n" , *(ary_p + (1 * 3)) ,
                  *(ary_p + (1 * 3)+ 1) , *(ary_p + (1 * 3) + 2));
        printf("添え字指定 = %d,%d,%d" , ary[1][0] , ary[1][1] , ary[1][2]);
        return 0;
}
```

ary[1]行目の各要素を呼び出しています
間接参照で得られる結果は、添え字指定でやっていることと同じことです
ご覧のとおりややこしいので、これはあまり使われません

しかし、「知っているが使わない」と「わからないから使わない」では大違いです
自分でソースを作って、いろいろなパターンを研究してみてください