ソフトゼミ A 第 7 回

ポインタ

今回はポインタ、アドレスという二つの概念を学習します。難しいですが、頑張っていきましょう。

◆ アドレス

a という変数を宣言したとき、コンピュータ上では、その変数”a”の値を保持するためにメモリの一部を使用しています。変数に割り当てられたメモリ上の位置を「アドレス」と呼びます。変数の住所といえばわかりやすいでしょう。int 型の変数 a、b が宣言されたときのメモリの割り当てのイメージは以下のようになります。

中身		a		b	
アドレス	[0214]	[0218](&a)	[0222]	[0226](&b)	[0230]

このようにメモリ上の適当な位置に変数が割り当てられ、アドレスを持ちます。割り当ての場所やアドレスは実行のたびに毎回変動するので覚えておきましょう。アドレスをプログラム上で利用したいときは、先頭にアドレス演算子&をつけて&a のようにして呼び出します。この例では、&a=0218 となります。scanf(“%d”,&a);なんかもアドレスを利用した関数です。詳しくは後ほど説明します。

◆ ポインタ

ポインタとは、先程解説したアドレスを保存することができる変数のことです。ポインタを使用することでアドレスを活用したプログラムを作成することができます。
宣言は以下の通りです。

```
データ型　*変数名(ポインタ名);
int *p;
double *q;
```

変数名の前に*をつけることでポインタ宣言となり、上記例では int 型のポインタ変数 p と double 型のポインタ変数 q が宣言されたことになります。*は変数名に含まれないことに注意してください。また、int、double とわざわざ型を分けています。これは変数の型によって保持に必要なメモリの量が異なるからです。次のページにポインタを用いたサンプルコードとその実行結果を示します。

```
#include<stdio.h>
int main(void) {
        int x, y, *p;
        x = 3;
        y = 5;
        p = &x;
        printf("case1   : x=3; y=5; p=&x;¥n");
        printf("result1 : x = %d, &x = %p, y = %d, &y = %p, p = %p, *p =  %d¥n", x, &x,
y, &y, p, *p);

        p = &y;
        printf("case2   : x=3; y=5; p=&y;¥n");
        printf("result2 : x = %d, &x = %p, y = %d, &y = %p, p = %p, *p =  %d¥n", x, &x,
y, &y, p, *p);

        *p = 7;
        printf("case3   : x=3; y=?; p=&y;¥n");
        printf("result3 : x = %d, &x = %p, y = %d, &y = %p, p = %p, *p =  %d¥n", x, &x,
y, &y, p, *p);
}
```

実行結果

```
case1    : x=3; y=5; p=&x;
result1 : x = 3, &x = 00AFFBEC, y = 5, &y = 00AFFBE0, p = 00AFFBEC, *p =  3
case2    : x=3; y=5; p=&y;
result2 : x = 3, &x = 00AFFBEC, y = 5, &y = 00AFFBE0, p = 00AFFBE0, *p =  5
case3    : x=3; y=?; p=&y;
result3 : x = 3, &x = 00AFFBEC, y = 7, &y = 00AFFBE0, p = 00AFFBE0, *p =  7
続行するには何かキーを押してください ...
```

case1 では p=&x;で p に x のアドレスを代入させています。そして、宣言時でもないのに *p という表現を使用している場面が出てきました。これは、「ポインタ変数 p が指しているアドレスの中身」を参照するときの表現です。case1 では p は x のアドレスを指しており、その中身を参照すると x の値である 3 となることがわかります。また、case2 では p に y のアドレスを代入したため、*p が参照する中身も y の値となります。そして、case3 では*p=7;

によって p が指す変数の中身を、すなわち変数 y の値を 7 に書き換えています。*p が参照する中身が 7 になっただけでなく、y の値も 7 となっていることがわかると思います。

アドレスを利用して変数の中身を書き換えられるので、main 関数から変数のアドレスを引数として関数に渡してしまえば、関数から直接 main 関数内の変数の値を書き換えることもできます。以下にその例を示します。

```
#include<stdio.h>
void twice(int *p) {
        *p *= 2;
}
int main(void) {
        int a;
        printf("a : ");
        scanf("%d", &a);

        printf("a = %d¥n", a);

        twice(&a);
        printf("a = %d¥n", a);
}
```

```
a : 3
a = 3
a = 6
続行するには何かキーを押してください ...
```

実行結果

main 関数で値を変化させていないうえ、twice 関数が void 型であるにも関わらず、a の値が入力した数値（ここでは 3）の 2 倍になっていることがわかります。それは、twice 関数の引数としてポインタ変数 p を指定し、main 関数で&a（a のアドレス）を twice 関数への引数として、twice 関数でメモリに保持された値を直接変更したからです。このコードでは scanf 関数を用いていますが、a=scanf…といった形でないにもかかわらず a に値が入っているのも同じ原理です。

◆ まとめ

以下にポインタの概念の簡単なまとめを示します。比較しながらサンプルコードの読み返しや練習問題を行い、理解を深めるとよいでしょう。

x : int 型（int x;)
p : int 型のポインタ(int *p;)
とする。

x ： 変数 x の『値』
&x ： 変数 x の『アドレス』

p = &x とすると、
p : 変数 p の『値』 = 変数 x の『アドレス』
*p : 『変数 p の値をアドレスとする変数の値』 = 変数 x の『値』

◆ 練習問題

1．大きさ 10 の配列を宣言し、そのアドレスを一行ずつ表示しなさい。

2. 2 つの変数のアドレスを受け取ってその中身の値を比較し、小さいほうの値を大きいほうの値に統一する関数。

void compare(int *p, int *q)

を作成しなさい。

ソフトゼミ∀第５回

ポインタ

ここではポインタの応用例などについて取扱います。より難解な内容になりますが覚えておいて損はないでしょう。

◆ 配列と関数、ポインタ

例えば、int a[3];のように大きさ３の配列を宣言したとします。このとき、a[0]、a[1]といった要素一つ一つにアドレスが割り当てられますが、それらはメモリ中の連続した位置に割り当てられます。このときのメモリの割り当てのイメージは以下のようになります。

中身		a[0]	a[1]	a[2]	
アドレス	[0214]	[0218](&a)	[0222]	[0226](&b)	[0230]

ここで、配列を関数に渡すことを考えてみましょう。例えば、void func(int a[])という関数があったとき、この関数を呼び出すには func(a);と引数には「配列名」を記述します。関数の引数として配列を渡すとき、「配列の各要素の値」ではなく「配列の先頭のアドレス」を渡しています。つまり、関数の中では「メモリに保持された値」を直接いじることになるので、関数内で配列の要素の値が変わってしまうと main 関数など他の場所でも連動して値が変わってしまうことになりますこれを避ける方法については第六回∀で取り扱いましたのでそちらを参照のこと)。以下に例を示します。

```
#include<stdio.h>
#define N 5
//このようにdefineを用いて宣言する文字列をマクロという。
//マクロはで定数を定義することができ、配列の要素数としても使用可能。
void array_twice(int a[], int n) {
        //nは配列の要素数とする。
        int i;
        for (i = 0; i < n; i++)a[i] *= 2;
}
int main(void) {
        int a[N];
        int i;
```

（次のページへ続く）

```
        for (i = 0; i < N; i++) {
                a[i] = i;
                printf("a[%d] = %d¥n", i, a[i]);
        }
        array_twice(a, N);
        printf("¥n");
        for (i = 0; i < N; i++ )printf("a[%d] = %d¥n", i, a[i]);
}
```

実行結果

```
a[0] = 0
a[1] = 1
a[2] = 2
a[3] = 3
a[4] = 4

a[0] = 0
a[1] = 2
a[2] = 4
a[3] = 6
a[4] = 8
続行するには何かキーを押してください ...
```

確かに、配列の中身が書き換わっていることがわかります。この array_twice 関数をポインタを用いて表現すると以下のようになります。

```
void array_twice(int *a, int n) {
        //nは配列の要素数とする。
        int i;
        for (i = 0; i < n; i++)*a++ *= 2;
}
```

ここで、関数の引数として配列を渡すときに「配列の先頭のアドレス」を渡すこと、配列の各要素のアドレスは連続であることを考えると、ひとつ次のアドレスを参照することで a[1]、a[2]と次の要素を参照できることになります。先ほどの例の array_twice 関数をこ

ちらに差し替えても問題なく動作します。こちらでは、先頭のアドレスを渡した後は*a++によってポインタの示す場所を動かしながら中身の値を変えていっています。

◆ リスト

ここで、構造体におけるポインタの概念を考えてみましょう。

```
struct cell {
        int value;
        struct cell *next;
};
```

このように cell 型の構造体が定義されているとして、main 関数において cell 型の構造体のポインタ p が struct cell *p;と宣言されたとすると、p が指す構造体の中身は(*p).value、(*p).next、または矢印のような表現を用いて p->value、p->next で参照することができます。注目するべきは next で、これは「cell 型の構造体のポインタ」です。つまり、「別の同じ cell 型を持つ別の構造体のアドレス」をメンバに含むことができ、これによって cell 型の構造体同士の連結が実現できるというわけです。このようなデータ構造をリストといいますが、とりあえずは例を見てみましょう。

```
#include<stdio.h>
struct cell {
        int value;
        struct cell *next;
};
int main(void) {
        struct cell a, b, c, *p;
        a.value = 1;
        a.next = &b;

        b.value = 3;
        b.next = &c;

        c.value = 7;
        c.next = NULL;
        /* ∀2. */
```

（次のページに続く）

```
        p = &a;
        while (p != NULL) {
                printf("%d¥n", p->value);
                p = p->next;
        }
}
```

実行結果

```
1
3
7
続行するには何かキーを押してください ...
```

リストの終わりを示すときには、そのポインタがなにも指していないことを示すNULLを使います。cell型の構造体ポインタpに最初にaのアドレス&aを指させておき、a.valueの値を出力したら、pにaの次の構造体、すなわち構造体bのアドレス&bを指させ、b.valueの値を出力し…といった動作を行い最終的にpがNULLを指す、すなわちリストの終わりに来たら走査を終了しています。リストには、配列よりも途中への値の挿入や削除がしやすいメリットがあります。また、今回のように「同じ型の構造体のアドレス」をメンバに含む構造体を自己参照型構造体といいます。