第10章　「ポインタ」のまとめ

10.1　用語のまとめ

教 p.226

関数の引数　**（復習）** C 言語の関数呼出しは値渡しである。つまり、ふつうは関数の中で呼出し側の変数を変更することはできない。

Q 10.1.1　（復習）以下のプログラムの出力を答えよ。

```
1  #include <stdio.h>
2
3  void hoge1(int x) {
4      x = 1;
5  }
6
7  int main(void) {
8      int a = 0;
9      hoge1(a);
10     printf("a=%d\n", a);
11     return 0;
12 }
```

答: _

教 p.227

アドレス　C 言語の変数には（register 変数などの例外を除き）メモリ上のアドレス（address）がある。

教 p.228

アドレス演算子　単項 _（空欄 10.1.1）演算子を変数（や配列の要素など=演算子の左辺に置けるもの）に適用すると、そのアドレスが得られる。単項& 演算子はアドレス演算子とも呼ばれる。

アドレスを printf で出力するときの書式指定は %p である。

教 p.229

ポインタ　アドレスを変数に格納することができる。このようなアドレスを格納する変数をポインタという。ポインタ型の変数の宣言は * を変数名の前につける。

```
1   int sato = 178;
2   int *isako;
3   isako = &sato;
```

- isako は int 型のオブジェクトのアドレスを格納することができる。これを“isako は sato を指す”という。isako の型は int 型のオブジェクトを指

すためのポインタ型（“int へのポインタ型”）である。（*の部分が主で int が従である。）
なお、型の一部であることを強調するために*を型のほうにくっつけて、

```
int* isako;
```

のような書き方をすることもある。（意味はまったく変わらない。）

- ただし、ポインタ変数宣言の*はそれぞれの変数名の前につける必要がある。

```
int *isako,  hiroko;
  /* isakoはintへのポインタ型、hirokoはint型になる。 */
```

```
int* isako,  hiroko;
  /* intのほうに*をくっつけても同じ。
      isakoはintへのポインタ型、hirokoはint型になる。 */
```

```
int *isako, *hiroko;
  /* isakoも hirokoもintへのポインタ型になる。 */
```

教 p.231

間接演算子　ポインタの前に単項 * （空欄 10.1.2）演算子をつけると、それが指すオブジェクトそのもの（ここでは変数と思って良い）を表す。つまり、=演算子の左辺に使えば代入ができ、右辺に使えばその値を表す。単項 * 演算子は間接演算子とも呼ばれる。

```
*hiroko = 180;    /* hirorkoの指すオブジェクトに180を代入 */
                  /* この例の場合 masaki = 180と同じ効果を持つ */
```

教 p.232

関数の引数としてのポインタ　関数の引数としてポインタを渡すと、次のようなことが可能になる。

- サイズの大きなデータをコピーせずに受渡しする。
- 呼出し側の変数の値を書き換える。
  （C 言語の関数の引数は値渡しなので、ポインタを使わなければ、呼出し側の変数の値を書き換えることはできない。）
- 複数の値を返す。

Q 10.1.2 以下のプログラムの出力を答えよ。

```
1  #include <stdio.h>
2
3  void hoge2(int *p) {
4      *p = 1;
5  }
6
7  int main(void) {
```

```
 8      int a = 0;
 9      hoge2(&a);
10      printf("a=%d\n", a);
11      return 0;
12  }
```

答: _

教 p.216

scanf 関数とポインタ　scanf 関数で int 型（%d に対応）や、double 型（%lf に対応）を受取るとき、変数の前に & をつけたのも、関数から呼出し側の変数の値を変更するためである。

教 p.238

ポインタの型　ポインタの型は、それが指すオブジェクトの型と正しく対応しなければならない。

この点は、ポインタのインクリメント・デクリメント（後述）のとき、さらに重要になる。

教 p.240

ポインタと配列　ポインタは配列の要素を指すこともできる。

```
1      int vc[5] = { 10, 20, 30, 40, 50 };
2      int *ptr = &vc[0];
```

&vc[0] で配列の先頭要素のアドレスを表す。（p.177 演算子の一覧表参照）

注意:

- int *ptr = &vc[0]; のようにポインタ変数の宣言と初期化を一度に行う宣言は、

```
1      int *ptr;
2      ptr = &vc[0];
```

と同じ意味になる。

```
1      int *ptr;
2      *ptr = &vc[0];    /* 間違い */
```

ではないので、注意が必要である。

教 p.241

- ptr + i は、（配列の要素の型にかかわらず）ptr が指す要素の i 個後ろの要素を指すポインタになる。
- * (ptr + i) は ptr[i] と書くこともできる。（2 つの書き方は C ではまったく等価である。）
- [] を伴わず現れた配列名は、配列の先頭要素へのポインタと見なされる。（ただし、sizeof のパラメータとなるときなど、いくつかの例外がある。）
  - scanf で文字列（%s に対応）を受取るために char の配列を渡すとき、&を付けない（教科書 p.212）のは、このためである。

– ポインタ同士の比較（==, <, >など）や減算（-）も可能である。

Q 10.1.3 以下のプログラムの出力を答えよ。

```
1  #include <stdio.h>
2
3  int main(void) {
4      int vc[] = { 2, 3, 5, 7, 11 };
5      int *p = vc;
6      printf("%d\n", *p);
7      return 0;
8  }
```

答: _

教 p.255

ポインタのインクリメント・デクリメント　配列の要素を指すポインタをインクリメント（++）・デクリメント（--）すると、それぞれ配列の次の要素・前の要素を指すようになる。

Q 10.1.4 以下のプログラムの出力を答えよ。

1.
```
1  #include <stdio.h>
2
3  int main(void) {
4      int vc[] = { 2, 5, 8, 11 };
5      int *p = vc;
6      p++;
7      printf("%d\n", *p);
8      return 0;
9  }
```

答: _

2.
```
1  #include <stdio.h>
2
3  int main(void) {
4      int vc[] = { 2, 5, 8, 11 };
5      int *p = vc;
6      (*p)++;
7      printf("%d\n", *p);
8      return 0;
9  }
```

答: _

教 p.244

配列の受渡し　関数の仮引数の宣言としては、`int vc[]` と `int *vc` はまったく同一である。このプログラムの関数 `int_set` の定義は、

```
1  void int_set (int *vc, int no, int val) {
2       ...
3  }
```

と書いてもまったく意味は同じである。

ポインタ使用上の注意

問 10.1.5 以下のプログラムはどこが間違っているか？

1. ```
   1  #include  <stdio.h>
   2
   3  /*--- n1とn2の和・差をsumとdiffに格納 ---*/
   4  void sum_diff(int n1, int n2, int *sum, int *diff) {
   5      *sum  = n1 + n2;
   6      *diff = (n1 > n2) ? n1 - n2 : n2 - n1;
   7  }
   8
   9  int main(void) {
   10     int  na, nb;
   11     int *wa, *sa;  /* ← ここを変えた */
   12
   13     puts("二つの整数を入力してください。");
   14     printf("整数Ａ：");   scanf("%d", &na);
   15     printf("整数Ｂ：");   scanf("%d", &nb);
   16
   17     sum_diff(na, nb, wa, sa);
   18
   19     printf("和は%dです。\n差は%dです。\n", *wa, *sa);
   20
   21     return (0);
   22 }
   ```

2. ```
   1  #include  <stdio.h>
   2
   3  int *sum_diff(int n1, int n2) {
   4      int ret[2];
   5
   6      ret[0]  = n1 + n2;
   7      ret[1]  = (n1 > n2) ? n1 - n2 : n2 - n1;
   8
   9      return &ret[0];  /* 先頭要素へのポインタを返す */
   10 }
   11
   12 int main(void) {
   13     int  na, nb;
   14     int *wasa;
   ```

```
15
16      puts("二つの整数を入力してください。");
17      printf("整数Ａ：");    scanf("%d", &na);
18      printf("整数Ｂ：");    scanf("%d", &nb);
19
20      wasa = sum_diff(na, nb);
21
22      printf("和は%dです。\n差は%dです。\n", wasa[0], wasa[1]);
23
24      return (0);
25  }
```

大抵の場合、問題がないように実行されてしまうので、こういう間違いはすぐに見つかる間違いよりタチが悪い。