ポインタ

ポインタとは「変数のアドレスを記憶することができる変数」とみなすことができる（注１参照）。Ｃ言語の特徴にポインタが使用できることがあげられるが、このポインタの取り扱いはＣ言語を習得する上での一つの壁として存在している。しかし、順を追ってきちんと学習していけば、必ず理解できるようになるので努力してもらいたい。まずは基本的なポインタの概念をつかむための例題を示すが、これだけで使いこなせるようになることは難しい。使いこなすためには多くのサンプルプログラムを見て、一つずつの処理の流れを理解していくことが大事である。

注１：正確には「データオブジェクトの配置されているメモリ上の先頭アドレスを記憶することができる変数である」と定義できる。データオブジェクトとは変数や後で習う構造体などが含まれるが、今の段階では変数とみなして話を聞いて良い。

変数とアドレス

ポインタについて理解するには「アドレス」とは何かをまず理解する必要がある。例えば下図の例のように char a = 'A'; と宣言した場合には、「メモリ上のある番地に変数 a としての領域を確保し、その領域に 'A' を格納する」ということになる。このある番地というのはコンパイラが自動的に決定する。下図の例では char 型の変数 a は 0x1000 番地（0x を頭に付けると 16 進数）に割り振られている様子を示している。変数のサイズは char 型は 1 バイト、int 型と float 型は 4 バイト、double 型 8 バイトとなる（環境によってはサイズが異なる）。例えば int 型の変数 c について確認すると、0x1005 番地から 0x1009 番地までの 4 バイト分の領域が確保されている。通常、「変数のアドレス」と言う場合には「変数が格納された領域の先頭のアドレス」を意味する。変数 c の例でいうと 0x1005 番地である（注２参照）。

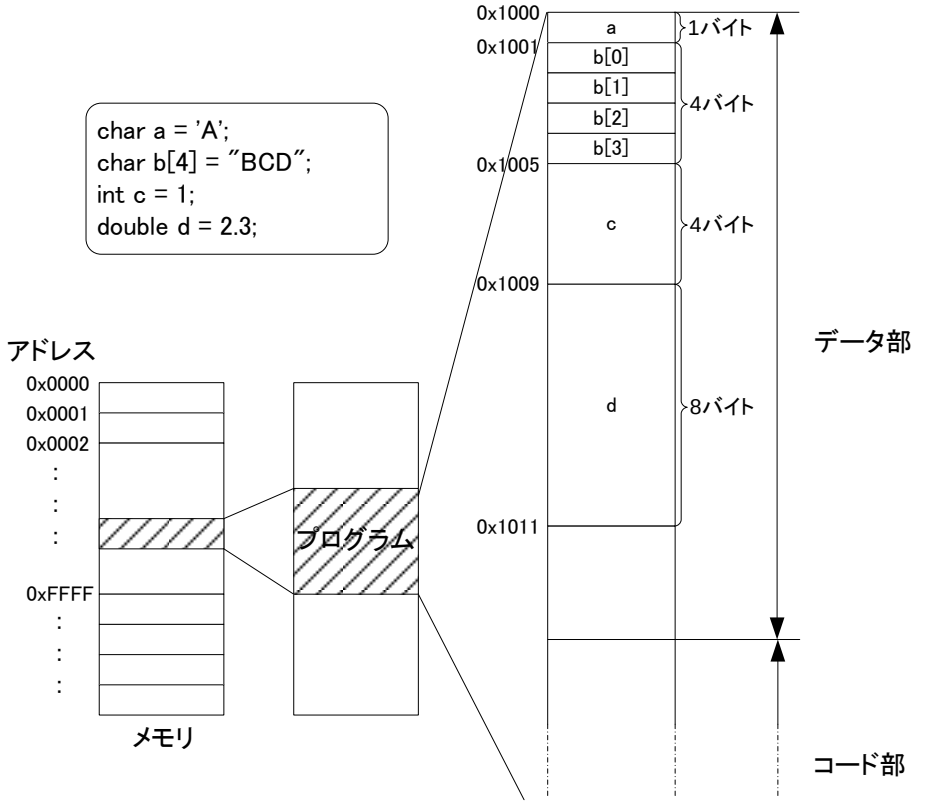

図１　メモリイメージ

また、コンパイラによって自動的に割り振られる変数のアドレスを知りたい場合にはアドレス演算子である「&」を用いる。変数aのアドレスを知りたい場合には「&a」とすればよい。図１の場合には0x1000が得られることになる。

注２：図１のメモリイメージでは隙間無く変数の領域が確保されている図を示したが、実際の割り当てはコンパイラによって決まるため、未使用の領域が発生する場合もある。また、バイト単位で図を示したが、C 言語自体はビットフィールドというものが用意されているためビット単位でデータを扱うこともできる。ビットフィールドは通常のソフトウェアを作成する場合には利用する機会はほぼ無いが、使用するメモリを節約したい時やマイコンを扱う場合などに用いると便利である。

ポインタ変数

ポインタ変数は「変数のアドレスを記憶することができる変数」と説明した。図２ではポインタ変数を char *pa;として宣言している。ポインタ変数も変数であるので、メモリ上に配置されることになる。また、pa = &a;と記述し、変数 a のアドレスをポインタ変数 pa の値として格納している様子を示している。この例の場合には 0x1000 が格納される。この状態を「ポインタ変数 pa は変数 a を指す」という。

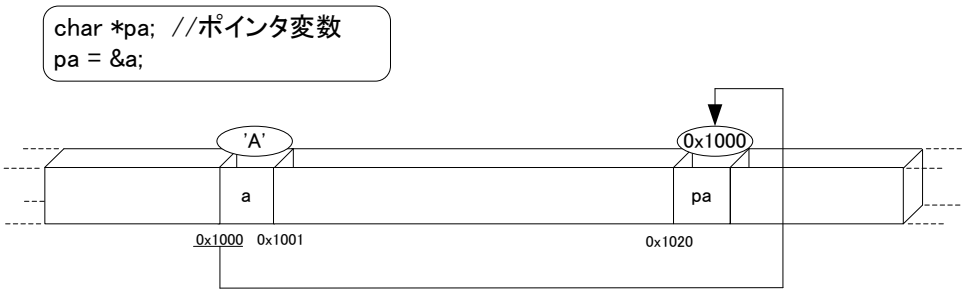

図２　ポインタ変数

変数とポインタ変数を用いた使い方

ポインタは必ず、

①ポインタ変数の宣言
②値（アドレス）の設定
③使用

の 3 ステップで用いる。次の使用例をよく見て、この 3 ステップを確認すること。間違いやすいのは「*」の使い方である。各ステップで「*」がどのように使われているかよく注意して確認すること。

プログラム例１

```
#include <stdio.h>

int main(void)
{
    int a, b;
    int *pa;                    ①

    a = 100;
    pa = &a;                    ②

    b = *pa + 1;                ③

    printf("a = %d¥n", a);
    printf("b = %d¥n", b);

    return 0;
}
```

実行結果

```
a = 100
b = 101
```

それでは各ステップを個別に説明していく。

①ポインタ変数の宣言

ポインタを使うための宣言を行う。宣言方法は以下の通り。

書式
（アドレスを記憶する変数の）データ型　*ポインタ変数名;

使用例のように宣言した場合には図３のようなメモリイメージとなる。

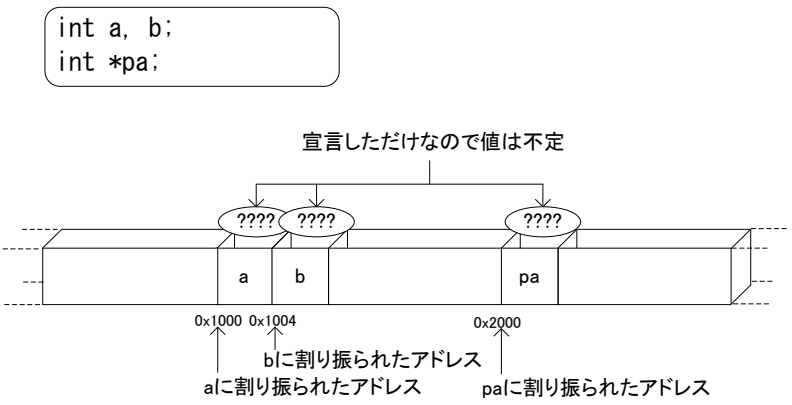

図３　変数とポインタ変数の宣言

②値（アドレス）の設定

宣言を行ったポインタ変数に値を代入する。

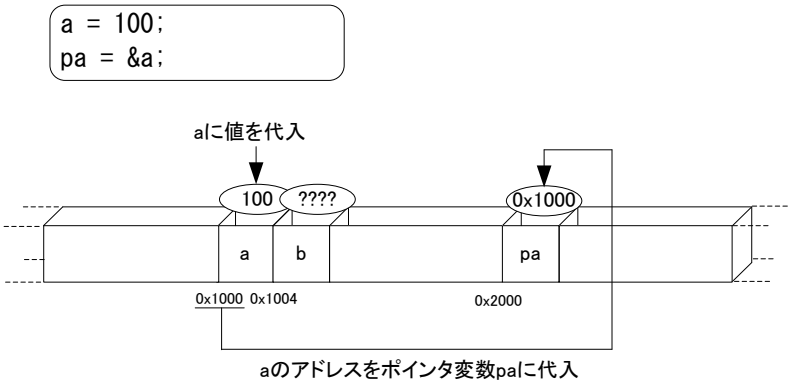

図４　アドレスの設定

③使用

宣言を行ったポインタ変数の頭に*を付けるとそのポインタ変数が指す変数の値を得ることができる。ここでの場合にはポインタ変数は pa であるので、*pa と書くことで変数 a の値を得ることができる。使用例で示したプログラムでは b = *pa+1;と記述している。この場合は b = a+1;と等しい結果が得られる。つまり 100+1 という演算が行われ、結果として 101 が変数 b に格納される。

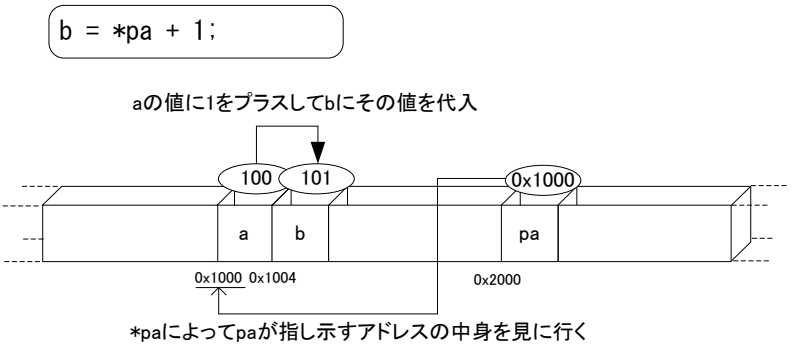

図５　ポインタ変数の使用方法

以下のプログラム例は変数のアドレス、変数の値、ポインタの値、ポインタが指す値を確認するプログラムである。printf 文の中で使用している%p はアドレスを表示するための変換仕様である。また、実行結果例の波線部分はコンパイラによって自動的に割り振られる値であるので、環境によって異なる結果となる。

プログラム例２

```
#include <stdio.h>

int main(void)
{
        int a = 256, *pa;

        pa = &a;

        printf( "変数 a のアドレス = %p¥n", &a );
```

```
        printf( "変数 a の値   = %d¥n", a );
        printf( "ポインタ pa の値 = %p¥n", pa );
        printf( "ポインタ pa の指す値 = %d¥n", *pa );

        return 0;
}
```

実行結果例

```
変数 a のアドレス = 0012FED4
変数 a の値   = 256
ポインタ pa の値 = 0012FED4
ポインタ pa の指す値 = 256
```

配列とアドレス

図６のようにプログラムを作成し、コンパイラによってアドレスが割り振られたものとする。配列変数、ポインタ変数の宣言方法は既に示しているので、ここでは配列のアドレスの格納方法を確認する。使い方は以下の書式に従う。

書式

ポインタ変数名 = 配列変数名

以下の例では pb = b;がこれにあたる。この場合には配列 b の先頭アドレスが代入されることになる。これは pb = &b[0];と書いた場合の処理に等しい。

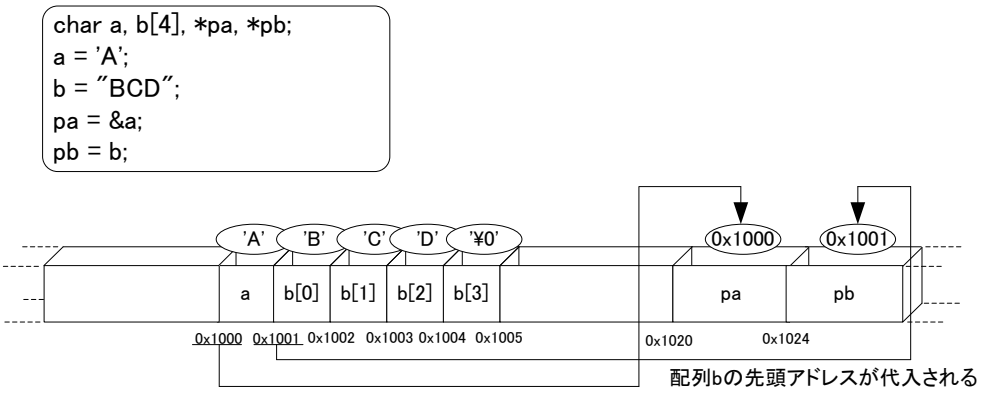

図６　配列とアドレス

図６ではポインタ変数のサイズを 4 バイトとして示している。各変数のサイズは以下のように sizeof 演算子を用いることで確認することが出来る。

プログラム例３

```
int main(void)
{
        printf("変数のサイズ¥n");
        printf("char   %d バイト¥n", sizeof(char));
        printf("int    %d バイト¥n", sizeof(int));
        printf("float  %d バイト¥n", sizeof(float));
```

```
        printf("double %d バイト¥n", sizeof(double));
        printf("ポインタのサイズ¥n");
        printf("char   %d バイト¥n", sizeof(char*));
        printf("int    %d バイト¥n", sizeof(int*));
        printf("float  %d バイト¥n", sizeof(float*));
        printf("double %d バイト¥n", sizeof(double*));

        return 0;
}
```

実行結果例

```
変数のサイズ
char   1 バイト
int    4 バイト
float  4 バイト
double 8 バイト
ポインタのサイズ
char   4 バイト
int    4 バイト
float  4 バイト
double 4 バイト
```

このように、ポインタ変数は指し示すデータ型がどんなものであっても同じサイズになることが分かる。

配列とポインタ変数を用いた使い方

ポインタの使用方法は「変数とポインタ変数を用いた使い方」で示した方法と基本的に同じであり、以下のように用いる。

①ポインタ変数の宣言
②値（アドレス）の設定
③使用

①および②については前節「配列とアドレス」で示した。ここでは③について具体的に説明していく。配列のアドレスをポインタ変数に代入した場合、配列の各要素はポインタ変数を使って参照することができる。この場合、**「ポインタ変数の値を変えずにデータを参照（相対的なポインタ参照）」**する方法と**「ポインタ変数の値そのものを更新してデータを参照（絶対的なポインタ参照）」**する方法の２通りがある。どちらを選んでもかまわないが、後者の場合には、ポインタ変数に格納されるアドレスが更新されるので、こちらの方の都合が良ければ、採用する。ただし、更新されたことを忘れていると、思わぬエラーの原因になるので、注意が必要である。

以下のプログラム例は char 型配列に文字列を格納し、相対的なポインタ参照と絶対的なポインタ参照をそれぞれ利用して１文字ずつ文字を表示するプログラムである。

プログラム例４

```
#include <stdio.h>

int main(void)
{
  char a[4]="ABC";
  char *pa, *pb;                    ①

  pa = a;                           ②-1

  printf("%c¥n", *pa);
  printf("%c¥n", *(pa+1));          ③-1
  printf("%c¥n", *(pa+2));

  pb = a;                           ②-2

  printf("%c¥n", *pb);
  pb++;
  printf("%c¥n", *pb);              ③-2
  pb++;
  printf("%c¥n", *pb);

  return 0;
}
```

実行結果

```
A
B
C
A
B
C
```

このプログラムで、①と②－１が実行された後、以下のようにアドレスが割り振られたものとする。

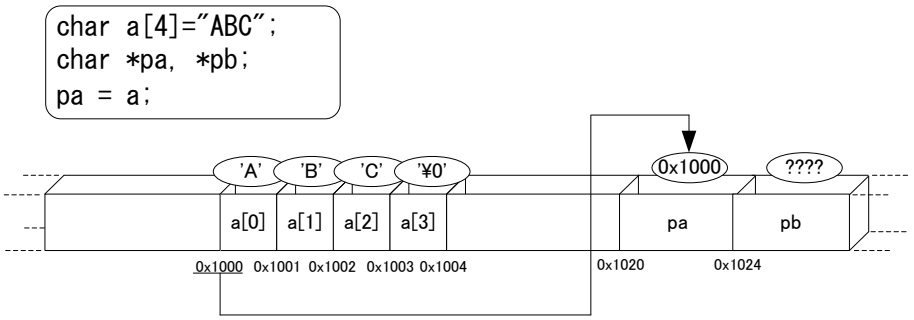

図７　ポインタと配列（１）

③－１の処理ではポインタ変数の値そのものは変えず、相対的なポインタ参照として、ポインタ変数に+1、+2 することで、配列の各要素を 1 文字ずつ出力している。ポインタ変数の加算は先に処理を行うことになるので、*(p+1)として、括弧をつける必要がある。

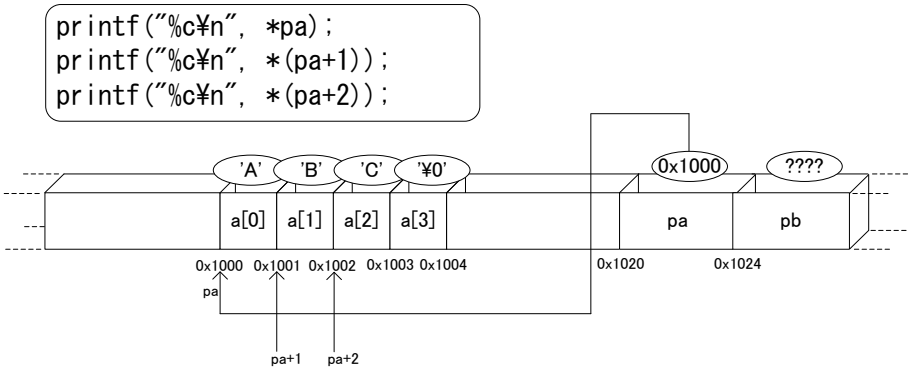

図８　ポインタと配列（２）

次に②－２が実行されると以下の図のようにポインタ変数 pb に配列 a のアドレスが格納される。

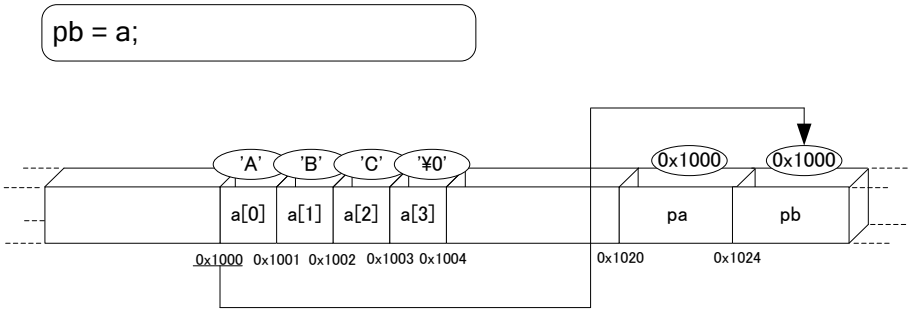

図９　ポインタと配列（３）

③－２の処理ではポインタ変数の値そのものを更新することで、絶対的なポインタ参照を行っている。pb は最初に配列 b の先頭アドレスである 0x1000 が格納されている。そのため*pb で出力すると、'A'が表示される。次に一つ目の pb++;でポインタ変数 pb の値は 0x1000 から 0x1001 に更新され、*pb で出力すると、'B'が表示される。同様にして二つ目の pb++;で 0x1001 から 0x1002 に更新され、*pb で出力すると、'C'が表示される。

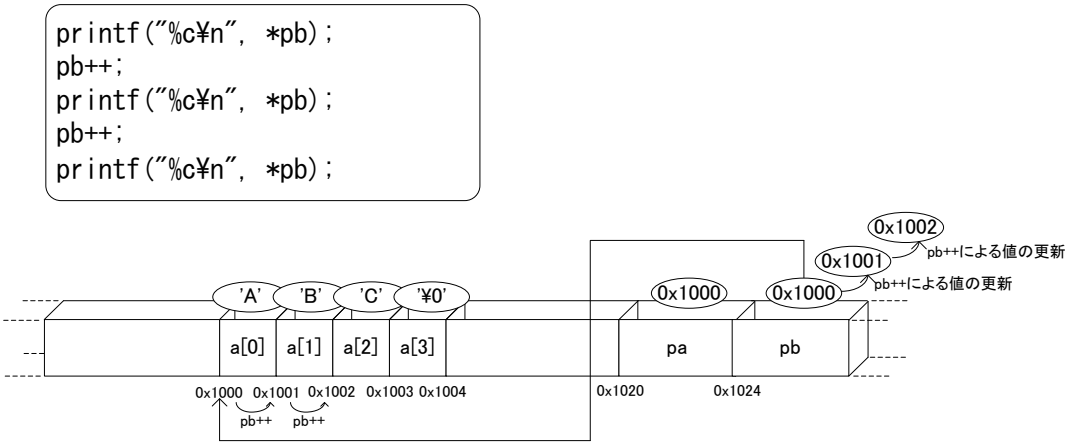

図１０　ポインタと配列（４）

ポインタのアドレス計算

前節では char 型の配列を例にとって説明したため、pa+1 や pb++によるアドレスの値の増分は 1 であった。しかし、int 型や double 型の配列を用いた場合には、アドレスの値の増分は異なってくる。「ポインタ変数に 1 加える」という処理は「ポインタ変数に格納されているアドレス + 1」ではなく、「ポインタ変数に格納されているアドレス + 型のサイズ」ということになる。つまり、「pb++」のような処理を行った場合には以下のように増える値が異なる。

char 型なら サイズは 1 バイトなので 1
int 型なら サイズ は 4 バイトなので 4
float 型なら サイズ は 4 バイトなので 4
double 型なら サイズ は 8 バイトなので 8

このように、型のサイズに応じて自動的に増える値が決まるため、プログラマは型の種類によって処理の方法を変えることなく、効率よくプログラムを作成できる。

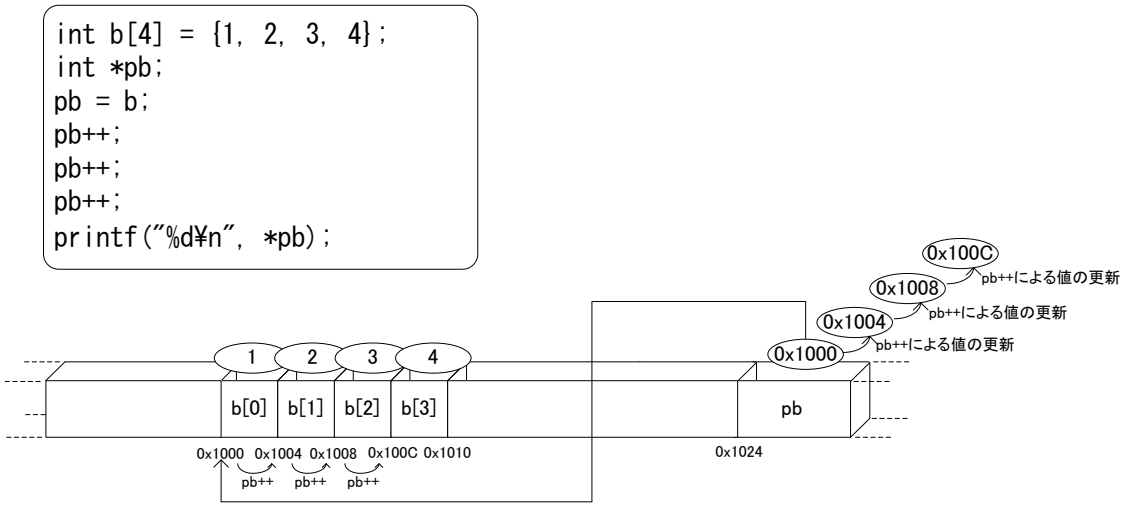

図１０　ポインタのアドレス計算：配列変数を int 型とすると各要素は 4 バイトのサイズをもっている。ポインタ変数をインクリメントした場合には各要素単位で参照が行われるようにアドレスの増分が自動的に決定される。

以下にポインタ変数のインクリメントによるアドレスの増分を確認するプログラムを示しておく。

プログラム例５

```
#include <stdio.h>

int main( void )
{
        char a[] = "Test";
        char *pa;
```

```
        pa = a;
        while ( *pa !='¥0' )  // NULL文字が見つかるまで繰り返す
        {
                printf("%p¥t%c¥n", pa, *pa);
                pa++;
        }

        return 0;
}
```

実行結果例

```
0012FED0     T
0012FED1     e
0012FED2     s
0012FED3     t
```

ポインタと文字列

通常、ポインタを用いる場合には、プログラム例５のようにして、配列を宣言した上で、そのアドレスをポインタ変数に代入して用いる。しかし、文字列の場合には、配列を使わずにメモリ上に領域を確保された文字列のアドレスを直接ポインタに指定することができる。以下は配列を使わずにポインタだけで文字列を表示するプログラムである。

プログラム例６

```
#include <stdio.h>

int main( void )
{
        char *pa;
        pa = "Test";
        printf( "%s¥n", pa );

        return 0;
}
```

実行結果

```
Test
```

ポインタ変数の宣言と代入の処理は以下のように初期化（宣言と同時に値を代入する）で記述することもできる。

```
        char *pa = "Test";
```

"Test" のように " " で囲んだ文字列を「文字列リテラル」と呼んだ。文字列リテラルは、基本的には値を変えない。この文字列リテラルはプログラムで使用するとメモリ上に領域が自動的に確保される。そのため、ポインタ変数にそのアドレスを格納することでプログラム例６のような使い方が可能となる。

しかし、値を変えないはずの文字列リテラルは、ポインタを用いた場合に変更が可能になってし

まう。例えば、

```
char *pa = "ABC";
strcpy(pa, "DEF");  // 文字列をコピーする関数
```

とした場合には、処理系によっては文字列の書き換えが可能になる。(処理系によっては動作しない。) 文字列リテラル内の文字列を変更してよいかどうかは、処理系に依存するが、C 言語で採用している ANSI 規格では、文字列リテラルの変更は定義されていない。そのため、書き換える必要のある文字列は、文字型配列に格納してから処理をするのが良い。

```
char a[] = "ABC";
strcpy(a, "DEF");
```

文字列とポインタに関する扱い方の練習として、次に文字列を逆順で表示するプログラムを示しておく。

プログラム例７

```
#include <stdio.h>

int main( void )
{
        char *pa, *pb;
        pa = pb = "Test";

        while ( *pa != '¥0' )  // NULL 文字までポインタ変数の値をインクリメント
        {
                pa++;
        }
        while ( pa > pb )      // 配列の先頭まで逆順で表示していく
        {
                pa--;
                printf("%c", *pa );
        }
        printf("¥n");

        return 0;
}
```

実行結果

```
tseT
```

演習　プログラム例１からプログラム例７までを作成して、動作を確認しなさい。