11 構造体、共用体、typedef

11–1 typedef ---新しいデータ型を定義する機構---

C言語では、データの使い方に合わせてデータ型に分かり易い名前を付けることが出来る。

例11．1（新しいデータ型の定義）　宣言

```
typedef int CentiMeter, Meter, KiroMeter;
```

が行なわれていれば、以降ではデータ型intの別名として CentiMeter, Meter, KiroMeter という名前を用いて、プログラム中で

```
CentiMeter height;
KiroMeter  distance;
```

という風な書き方も出来る。

例11．2（新しいデータ型の定義） 宣言

```
typedef char *String;
```

が行なわれていれば、以降では

```
String s = "abc";
```

という宣言と

```
char *s = "abc";
```

注釈：
「char *String;」の中の「String」を変数名 s で置き換え、初期設定部をつなげた。

という宣言は同等になる。

⇨ 上の typedef 宣言は「char 型へのポインタ」の総称として String というデータ型名を使うことを宣言している。

---

typedef宣言の利点

- 変数宣言をコンパクトに行なうことが出来る。
- 使い方に合わせてうまくデータ型に名前を付ければ、プログラムが読み易くなる。
- プログラムの移植性向上に役立つ。

例題11．3（N次元ベクトル空間の世界）　一般に、行列 $\mathbf{A}$ と(縦)ベクトル $\vec{\mathbf{x}},\vec{\mathbf{y}}$ に対して $\mathbf{A}(\vec{\mathbf{x}}+\vec{\mathbf{y}})=\mathbf{A}\vec{\mathbf{x}}+\mathbf{A}\vec{\mathbf{y}}$ であるが、これを

$$\mathbf{A}=\begin{pmatrix} \mathbf{1.1} & \mathbf{-2.2} & \mathbf{3.3} \\ \mathbf{-4.4} & \mathbf{5.5} & \mathbf{-6.6} \\ \mathbf{7.7} & \mathbf{-8.8} & \mathbf{9.9} \end{pmatrix}, \quad \vec{\mathbf{x}}=\begin{pmatrix} \mathbf{1.0} \\ \mathbf{2.0} \\ \mathbf{3.0} \end{pmatrix}, \quad \vec{\mathbf{y}}=\begin{pmatrix} \mathbf{-0.1} \\ \mathbf{0.2} \\ \mathbf{-0.3} \end{pmatrix}$$

に対して確認するプログラムを作成せよ。 但し、新しいデータ型を定義したり、機能分割を適切に行ったりすることによって、プログラムは出来るだけ読み易く構成せよ。

(考え方)　ここで出てくる基本的なデータは3次元ベクトルと3×3行列であるので、これらのためのデータ型にデータの意味を表す名前を付けると、プログラムが読み易くなる。そこで、例えば...

```
#define N 3
typedef double Scalar;
typedef Scalar Vector[N];
typedef Scalar Matrix[N][N];
```

機能分割に関しては、A($\vec{x}$+$\vec{y}$) と A$\vec{x}$+A$\vec{y}$ の計算をして各々の計算結果を出力する必要があることを考慮に入れなければならない。

ベクトルの加算や
線形変換の計算,
計算結果(ベクトル)の表示

をmain()関数の中で見通し良く記述するためには、これらの基本的な処理を例えば関数として定義したり、引数付きマクロとして定義すればよい。

（プログラミング） 問題の中で使われた名前に出来るだけ合わせるために、ベクトル $\vec{x}$, $\vec{y}$, $\vec{x}+\vec{y}$, $A(\vec{x}+\vec{y})$, $A\vec{x}$, $A\vec{y}$, $A\vec{x}+A\vec{y}$ を表す変数として各々 x, y, x_y, Ax_y, Ax, Ay, Ax_Ay という名前の変数を用意してプログラムを構成した。

[motoki@x205a]$ nl typedef-vector-space.c

```
     1 #include <stdio.h>

     2 #define  N  3

     3 #define  Print(title, vector)                     \
     4              printf("%s\n", title);               \
     5              printf("   (%7.3f ", vector[0]);     \
     6              for (i=1; i<N; ++i)                  \
     7                printf("%7.3f ", vector[i]);       \
     8              printf(")\n")

     9 typedef  double  Scalar;
```

```
10 typedef  Scalar  Vector[N];
11 typedef  Scalar  Matrix[N][N];

12 void add(Vector x, Vector y, Vector z);          /* x=y
13 void linear_trans(Vector x, Matrix A, Vector y); /* x=A

14 int main(void)
15 {
16   int     i;
17   Vector  x = {1.0, 2.0, 3.0}, y = {-0.1, 0.2, -0.3},
18           x_y, Ax_y, Ax, Ay, Ax_Ay;
19   Matrix  A = {{1.1, -2.2, 3.3},
20                {-4.4, 5.5, -6.6},  /* これらのA,x,yに *
21                {7.7, -8.8, 9.9}};  /* 関して A*(x+y)= *
22                                    /* A*x+A*y を確認  *
23   add(x_y, x, y);
24   linear_trans(Ax_y, A, x_y);
```

```
25    Print("A*(x+y) =", Ax_y);

26    linear_trans(Ax, A, x);
27    linear_trans(Ay, A, y);
28    add(Ax_Ay, Ax, Ay);
29    Print("A*x+A*y =", Ax_Ay);
30    return 0;
31 }

32 void add(Vector x, Vector y, Vector z)
33 {
34    int  i;

35    for (i=0; i<N; ++i)
36      x[i] = y[i]+z[i];
37 }
```

```
38 void linear_trans(Vector x, Matrix A, Vector y)
39 {
40   int  i, k;

41   for (i=0; i<N; ++i) {
42     x[i] = 0.0;
43     for (k=0; k<N; ++k)
44       x[i] += A[i][k] * y[k];
45   }
46 }
[motoki@x205a]$  gcc typedef-vector-space.c
[motoki@x205a]$  ./a.out
A*(x+y) =
   (  5.060  -9.680  14.300 )
A*x+A*y =
   (  5.060  -9.680  14.300 )
[motoki@x205a]$
```

11–2 構造体の定義

Pascal や Fortran90 等の他の (命令型) プログラミング言語と同様に、C 言語においても、関連するデータを 1 つにまとめて扱うことが出来る。

C 言語の場合は、

「関連するデータを 1 つにまとめたもの」を構造体

と呼ぶ。また、

構造体の構成要素をメンバ、

メンバを区別するための名前をメンバ名

という。

例11．4（構造体の宣言；トランプのカード） トランプのカードを識別するためのデータ構造

pips (int) ··· **1～13**
suit (char) ··· 's', 'h', 'd', 'c'

を持った変数 c1, c2 は次のように宣言することが出来る。

(方法1) 直接定義する。

```
struct {
  int  pips;
  char suit;
} c1, c2;
```

毎回長いのを書かないといけない。

(方法2) まず構造体の形に名前(タグという)を付けてから、...。

```
struct card {
  int  pips;
  char suit;
};
struct card  c1, c2;
```

「struct card」をデータ型の名前として使うことが出来る。

(方法3) まず構造体の形に名前付け、さらに、それに新しいデータ型としての名前を付けてから、....

```
struct card {
  int  pips;
  char suit;
};
typedef  struct card  Card;
Card  c1, c2;
```

「struct card」と「Card」をデータ型の名前として使うことが出来る。

(方法4) 構造体の形に新しいデータ型としての名前を付けてから、....

```
typedef struct {
  int  pips;
  char suit;
} Card;
Card  c1, c2;
```

「Card」をデータ型の名前として使うことが出来る。

構造体はいくらでも複雑に出来る。

例えば、

- 配列や構造体をメンバに出来る。
- 構造体の配列も許される。(例えば、下の例11.7)

11–3 構造体メンバへのアクセス

- 構造体メンバへのアクセスの仕方は次の2つ。

  構造体変数 . メンバ名

  構造体へのポインタ -> メンバ名

  ・・・ 次のものと同等。
  (* 構造体へのポインタ) . メンバ名

- 計算機内部では . も -> も演算子 として扱われる。
  (メンバアクセス演算子という。)

例11．5（構造体要素へのアクセス；トランプのカード） 先の例 **11.4** の様に変数 c1, c2 が宣言されていた場合、例えば

```
c1.pips = 3;
c1.suit = 's';
c2 = c1;
```

により、スペードの**3**を表すコードが**2**つの変数 c1, c2 にセットされる。

例11．6（配列を構成要素とする構造体）　構造体

```
struct person {
  int id;
  char name[40];
  long phone;
};
```

に関して、次の様なアクセスが可能。

構造体変数.name[5]

↑

こちらの方が強い。

（. も [] も最高の優先順位を持つが、
この中では左側のものが優先される。）

11–4 演算子の優先順位と結合性：まとめ

優先順位高

<table>
<tr><th>演算子</th><th>結合性</th></tr>
<tr><td>関数の引数をくくる丸括弧（ ） 配列添字をくくる四角括弧 [ ]
メンバアクセス演算子 -> .</td><td>左から右</td></tr>
<tr><td>+（単項）-（単項） ++ -- sizeof( ) ! キャスト
ビット反転~ 間接演算子* 番地演算子&</td><td>右から左</td></tr>
<tr><td>* / %</td><td>左から右</td></tr>
<tr><td>+ -</td><td>左から右</td></tr>
<tr><td>左シフト<< 右シフト>></td><td>左から右</td></tr>
<tr><td>< <= > >=</td><td>左から右</td></tr>
<tr><td>== !=</td><td>左から右</td></tr>
<tr><td>ビット積&</td><td>左から右</td></tr>
<tr><td>ビット排他的和^</td><td>左から右</td></tr>
<tr><td>ビット和|</td><td>左から右</td></tr>
<tr><td>&&</td><td>左から右</td></tr>
<tr><td>||</td><td>左から右</td></tr>
<tr><td>条件演算子 条件 ? 式 : 式</td><td>右から左</td></tr>
<tr><td>= += -= *= /=</td><td>右から左</td></tr>
<tr><td>コンマ演算子,</td><td>左から右</td></tr>
</table>

11–5 例題：複素多項式の計算

複素数についての復習

- 純虚数 $\sqrt{-1}$ を i で表す。
- 複素数は次の様な形で表せる。この時の x を実部、 y を虚部という。

  x+iy

  (但し、x と y は実数、iは純虚数。)
- ２つの複素数 $z_1 = x_1 + iy_1$ と $z_2 = x_2 + iy_2$ の和は

$$z_1+z_2 = (x_1+iy_1)+(x_2+iy_2) = (x_1+x_2)+i(y_1+y_2)$$

- ２つの複素数 $z_1 = x_1 + iy_1$ と $z_2 = x_2 + iy_2$ の積は

$$z_1z_2 = (x_1+iy_1)(x_2+iy_2) = (x_1x_2-y_1y_2)+ i(x_1y_2+y_1x_2)$$

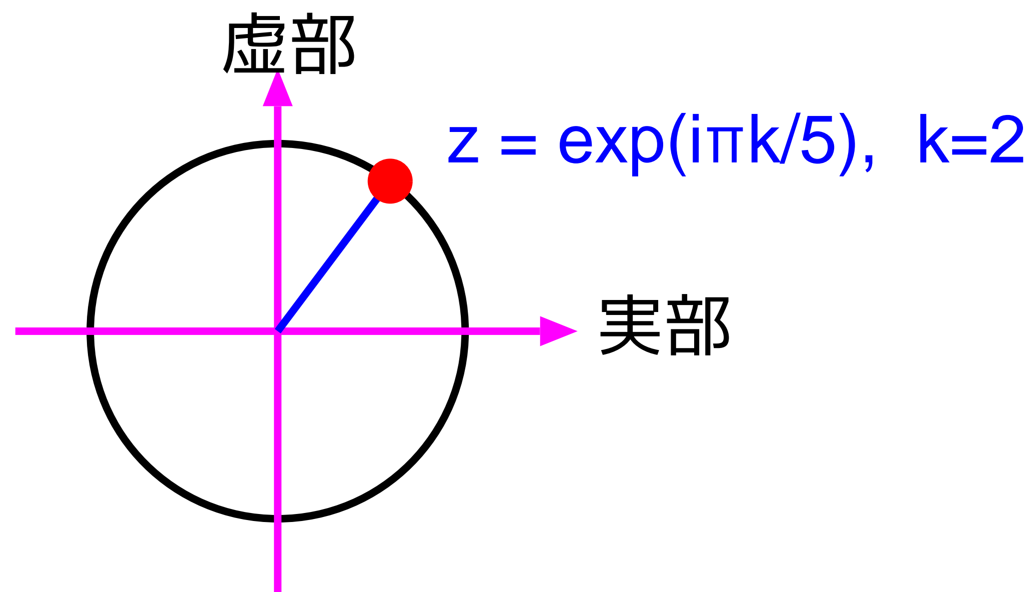
例題11．7（複素多項式の計算）　非負整数データ $\mathbf{n}$ と複素数データ $\mathbf{C_n, C_{n-1}, \ldots, C_0}$ を読み込み、これらの定数値の下で複素多項式

$$\mathbf{C_n z^n + C_{n-1} z^{n-1} + C_{n-2} z^{n-2} + \cdots + C_1 z + C_0}$$

の値が

$$\mathbf{z = e^{i\pi k/5} = \cos\frac{\pi k}{5} + i\sin\frac{\pi k}{5} \qquad (k=0,1,2,3,\ \ldots,\ 9)}$$

のそれぞれの値に対してどの様に変化するかを、表の形に見易く出力するCプログラムを作成せよ。



(設計方針)

- この問題の場合、複素数を基本的なデータとして扱うことが出来れば計算処理を実数型の多項式の計算と全く同じ様に進めることが出来る。
  ⟹複素数を構造体で表し、
  そのデータ型に Complex という名前を付ける。

- 多項式の係数 C_n, C_{n-1}, ..., C_0 を保持するために(Complex型)配列を用意するのが妥当である。
  この配列領域を宣言によって確保する場合その大きさはコンパイル時に確定してなければならないので、多項式の最大次数をマクロで設定して、実際の処理では確保した係数用の配列領域の一部だけを使う。

- 多項式の計算を効率的に行うために、(例えば「川合,(岩波講座ソフトウェア科学2) プログラミングの方法,岩波書店」p.145 ~ 146 で説明されている)Hornerの方法に従って
  $$(((C_n z+C_{n-1})z+C_{n-2})z+ \cdots +C_1)z+C_0$$
  という順序で計算する。

- 複素多項式の計算をプログラム上で実多項式の場合と全く同じ様に見通し良く記述したい。

⇨ 複素数同士の加算をして、その結果を値として返す関数 `sum_of()` と 複素数同士の乗算をして、その結果を値として返す関数 `product_of()` を用意する。

(プログラミング)

```
[motoki@x205a]$  nl struct-complex-polynomial.c
     1 /*********************************/
     2 /*                               */
     3 /* Horner法による複素多項式の計算 */
     4 /*                               */
     5 /*********************************/

     6 #include <stdio.h>
     7 #include <math.h>

     8 #define  MAX_DEGREE  100         /* 複素多項式の次数の...
     9 #define  PI  3.141592653589793 2 /* 円周率

    10 typedef struct{                /*--- 複素数の構造体 ---*/
    11     double re;                 /* 実部, real part      */
    12     double im;                 /* 虚部, imaginary part */
```

```
13 } Complex;

14 Complex sum_of(Complex z1, Complex z2);     /*引数の和.
15 Complex product_of(Complex z1, Complex z2); /*引数の積.

16 /*----------------------------------------------
17 /* 主プログラム
18 /*----------------------------------------------
19 /*    多項式の次数と複素係数を読み込み、
20 /*    色々な変数値 z に対して多項式の値を表の形に表示する...
21 /*----------------------------------------------
22 int main(void)
23 {
24   int degree, i, k;
25   Complex c[MAX_DEGREE+1], /* 多項式の係数を入れる配列
26           z,               /* 多項式の変数 z を入れる領.
27           result;          /* 計算結果を溜めていく領域
```

```
28   printf("Input the degree(<=100) of the polynomial: ")
29   scanf("%d", &degree);
30   for (i=degree; i>=0; --i) {
31     printf("  Input the real and the imaginary part"
32            " of the %d-th coefficient: ", i);
33     scanf("%lf %lf", &c[i].re, &c[i].im);
34   }

35   printf("\ndegree = %d\n", degree);
36   for (i=degree; i>=0; --i)
37     printf("  c[%d] = (%e)+(%e)i\n",
           i, c[i].re, c[i].im);
38   printf("\n k  %16sz%15s  c[d]*z^d+c[d-1]*z^(d-1)+ ...
39          "--  ------------------------------"
40          "  -----------------------------------\n
```

```
41   for (k=0; k<10; ++k) {
42     z.re = cos(PI*k/5);
43     z.im = sin(PI*k/5);
44     result = c[degree];
45     for (i=degree-1; i>=0; --i)
46       result = sum_of(product_of(result, z), c[i]);
47     printf("%2d  (%13.6e)+(%13.6e)i"
              "  (%13.6e)+(%13.6e)i\n",
48            k, z.re, z.im, result.re, result.im);
49   }
50   return 0;
51 }

52 /*-----------------------------------------------------
53 /* 複素数の和を求めて返す関数 sum_of
54 /*-----------------------------------------------------
55 /*   (仮引数) z1 : 複素数
```

```
56 /*              z2 : 複素数
57 /*   (関数値) : z1+z2 の値
58 /*---------------------------------------------------------
59 Complex sum_of(Complex z1, Complex z2)
60 {
61     Complex result;

62     result.re = z1.re+z2.re;
63     result.im = z1.im+z2.im;
64     return result;
65 }

66 /*---------------------------------------------------------
67 /* 複素数の積を求めて返す関数 product_of
68 /*---------------------------------------------------------
69 /*   (仮引数) z1 : 複素数
70 /*              z2 : 複素数
```

```
71 /*   (関数値) : z1*z2 の値
72 /*-----------------------------------------------
73 Complex product_of(Complex z1, Complex z2)
74 {
75      Complex result;

76      result.re = z1.re*z2.re - z1.im*z2.im;
77      result.im = z1.re*z2.im + z1.im*z2.re;
78      return result;                   修正
79 }
[motoki@x205a]$  gcc struct-complex-polynomial.c -lm
[motoki@x205a]$  ./a.out
Input the degree(<=100) of the polynomial:  3
  Input the real and the imaginary part of the 3-th coefficien

  Input the real and the imaginary part of the 2-th coefficien
```

```
  Input the real and the imaginary part of the 1-th coefficier

  Input the real and the imaginary part of the 0-th coefficier



degree = 3
  c[3] = (1.000000e+00)+(-2.000000e+00)i
  c[2] = (-3.000000e+00)+(4.000000e+00)i
  c[1] = (5.000000e+00)+(-6.000000e+00)i
  c[0] = (-7.000000e+00)+(8.000000e+00)i

 k                   z                     c[d]*z^d+c[d-1]*z^(d-1)+
--  --------------------------------   ----------------------------
 0  ( 1.000000e+00)+( 0.000000e+00)i   (-4.000000e+00)+( 4.0000
 1  ( 8.090170e-01)+( 5.877853e-01)i   (-2.566385e+00)+( 6.0368
 2  ( 3.090170e-01)+( 9.510565e-01)i   (-1.657253e+00)+( 6.9320
 3  (-3.090170e-01)+( 9.510565e-01)i   ( 1.572893e+00)+( 1.0930
```

```
 4  (-8.090170e-01)+( 5.877853e-01)i  (-2.430068e+00)+( 2.0215
 5  (-1.000000e+00)+( 1.224606e-16)i  (-1.600000e+01)+( 2.0000
 6  (-8.090170e-01)+(-5.877853e-01)i  (-2.089617e+01)+( 6.7289
 7  (-3.090170e-01)+(-9.510565e-01)i  (-1.219093e+01)+(-9.3085
 8  ( 3.090170e-01)+(-9.510565e-01)i  (-6.016509e+00)+( 2.1237
 9  ( 8.090170e-01)+(-5.877853e-01)i  (-5.815581e+00)+( 3.9631
[motoki@x205a]$
```

11–6 関数引数としての構造体

構造体を関数の引数として渡すことも、関数値として返すことも可能である。しかし、

- 構造体を関数引数として引数結合する際、および
- 構造体を関数値として変数に代入する際

には構造体が丸ごとコピーされることになる。

⇨ 大きな構造体の場合には、計算効率の低下を防ぐために、参照呼び出し(i.e.構造体へのポインタの受渡し)や引数付きマクロを使うべき。

11–7 自習 構造体の初期化

配列の場合と同様の書き方で構造体の初期化を行なうことが出来る。

例11．8（構造体の初期化）

```
typedef struct {
  int  pips;
  char suit;
} Card;
Card  c={13, 'h'};                ⇨ c.suit='h',  c.pips=13

typedef struct {
  char *name;
  int  calories;
}Fruit;
Fruit apple={"apple", 150};       ⇨ apple.name="apple",
                                    apple.calories=150
```

```
typedef struct {
  double re;
  double im;
} Complex;
Complex a[3][2] = {
  {{1.0, 2.0}, {3.0}},
  {{5.0, 6.0}, {7.0, 8.0}}
};
```

⇨ a[0][0].re=1.0, a[0][0].im=2.0,
a[0][1].re=3.0, a[0][1].im=0.0,

a[1][0].re=5.0, a[1][0].im=6.0,
a[1][1].re=7.0, a[1][1].im=8.0,

a[2][0].re=0.0, a[2][0].im=0.0,
a[2][1].re=0.0, a[2][1].im=0.0

(指定されなかった a[2] と a[0][1].im はゼロクリアされる。)

11–8 共用体

共用体：

- 色々な種類のデータを選択的に1つのデータ領域で表せる様にしたもの。
- 共用体定義や参照の構文は構造体の場合とほぼ同じ。
（キーワードが `struct` から `union` になっただけ。）
- 共用体の中のデータを正しく解釈して使うのはプログラマの責任。

例題11．9（共用体） 実数型データを `float` 型で読み込み、そのビット列を `int` 型と見て16進表示するCプログラムを作成せよ。

(考え方)　1つのデータ領域をfloat型と見たりint型と見たりする訳だから、このデータ領域を共用体として確保すれば良い。

i, f　(int/float型)　... ある時はint型のデータと見る / ある時はfloat型のデータと見る

また、
int型の値を16進表示するためには、
単に printf( ) の書式指定の中で %x という変換指定をしてやれば良い。

(プログラミング)

[motoki@x205a]$ nl struct-union-int-or-float.c Enter

```
 1 /*----------------------------------------------------*/
 2 /* 実数型データを float 型で読み込み、                  */
 3 /* そのビット列を int 型と見て16進表示するCプログラム */
 4 /*----------------------------------------------------*/

 5 #include <stdio.h>

 6 typedef union {
 7   int   i;
 8   float f;
 9 }Number;

10 int main(void)
11 {
12   Number  num;
```

```
13    scanf("%f", &num.f);
14    printf("Float number %g and int number %d are\n"
15        " commonly represented by a bit sequence %#.8x.\n",
16        num.f, num.i, num.i);
17    return 0;
18 }
```

```
[motoki@x205a]$ gcc struct-union-int-or-float.c  Enter
[motoki@x205a]$ ./a.out  Enter
1.0  Enter
Float number 1 and int number 1065353216 are
 commonly represented by a bit sequence 0x3f800000.
[motoki@x205a]$ ./a.out  Enter
1e-38  Enter
Float number 1e-38 and int number 7136238 are
 commonly represented by a bit sequence 0x006ce3ee.
[motoki@x205a]$ ./a.out  Enter
```

```
1e-45  Enter
Float number 1.4013e-45 and int number 1 are
 commonly represented by a bit sequence 0x00000001.
[motoki@x205a]$
```

この実行結果により、
同じビット列であっても、それがどういう内部表現方式に従っているかによって表されるデータが全く違うものになることが例示されている。

11–9 自習 ビットフィールド

- 構造体や共用体の中の`int`型,`unsigned`型のメンバは、ビット長を指定することが出来る。

（こういうメンバをビットフィールドという。）

⇨ コンパイラは、
それらを最小限のワードに詰め込む。

例題11．10（ビットフィールド） 実数型データを `float` 型で読み込み、そのビット列を2進表示するCプログラムを作成せよ。

(考え方)　読み込んだデータを構成するビット列をビット毎(あるいは数ビット毎)に調べることが出来れば、上の桁から順に調べてその結果に応じて対応する2進文字列を出力することが出来る。

そこで、読み込んだfloat型データを4ビット毎に調べられる様に、読み込んだデータを格納する変数をfloat型とint型の共用体として用意し、その共用体の中のint型メンバは更に細かく4ビットのビットフィールドの集まりとして構成する。

実際、この様にしておくと、4ビット毎にその(整数としての)値を調べることが出来るから、あとはその16種類の値の各々に対して16種類の文字列 0000, 0001, ..., 1111 の中の対応する文字列を出力するだけである。

(プログラミング)

```
[motoki@x205a]$ nl struct-union-bit-field.c
     1 /*-----------------------------------------
```

```
 2 /* 実数型データを float 型で読み込み、
 3 /* そのビット列を4bitの束の列と見て2進表示するCプログラ...
 4 /*------------------------------------------------------

 5 #include <stdio.h>

 6 typedef struct {
 7   unsigned  b0:4, b1:4, b2:4, b3:4,
                b4:4, b5:4, b6:4, b7:4;
 8 }Nibble_seq;

 9 typedef union {
10   float      f;          /* 4 byte を仮定 */
11   int        i;          /* 4 byte を仮定 */
12   Nibble_seq nibble;
13 }Number;
```

```
14 typedef  char  *String;

15 int main(void)
16 {
17   Number  num;
18   String  bit_seq[16] = {
19     "0000", "0001", "0010", "0011",
       "0100", "0101", "0110", "0111",
20     "1000", "1001", "1010", "1011",
       "1100", "1101", "1110", "1111"
21   };

22   scanf("%f", &num.f);
23   printf("Float number %g and int number %d are\n"
24          "commonly represented by a bit sequence\n"
25          "    %s%s %s%s %s%s %s%s.\n",
26          num.f, num.i,
```

```
27              bit_seq[num.nibble.b7],
                bit_seq[num.nibble.b6],
28              bit_seq[num.nibble.b5],
                bit_seq[num.nibble.b4],
29              bit_seq[num.nibble.b3],
                bit_seq[num.nibble.b2],
30              bit_seq[num.nibble.b1],
                bit_seq[num.nibble.b0]);
31    return 0;
32 }
[motoki@x205a]$  gcc struct-union-bit-field.c
[motoki@x205a]$  ./a.out
1.0
Float number 1 and int number 1065353216 are
commonly represented by a bit sequence
    00111111 10000000 00000000 00000000.
[motoki@x205a]$
```

– – –