11 基本的なデータ型と構造体, 共用体

各々の記憶領域に記録されたビット列がどういう内部表現方式に従っているかは、ビット列自身の中に明示的な情報として含まれている訳ではなく、その記憶領域を使う側 (プログラム側) が決めることであった。

⇨ プログラムはその中で使用される各々の変数や配列の中のデータがどういう内部表現方式に従うかの情報を含む。

C言語でこれらの情報を明示しているのは変数や配列の宣言で、

`int a;` と宣言すると
例えば32ビット, 負数は2の補数で整数を表す方式が、

`float a;` と宣言すると
例えば32ビット, IEEE規格754で実数を表す方式が、

`double a;` と宣言すると
例えば64ビット, IEEE規格754で実数を表す方式が、

変数 a の計算機内部の表現方式として想定される。

これらの、計算機内部のデータ表現方式に対応する

`int, float, double, ...`

といったものを、プログラム上ではデータ型と呼ぶ。

プログラミング言語によっては
複数のデータをまとめて1つの変数として扱う仕組みも用意されている。

⇨ この様な変数の内部構成もやはりプログラム内に宣言されていて、計算機内部のデータ表現の方式に対応するので、データ型と考える。

データ型の分類

- 基本型
  - 算術型
    - 整数型
      - char 型 ···8 ビットの整数を表せる。
        char, ...
      - int 型 ···[signed] int, unsigned [int], ...
      - 列挙型 ···実体は int の部分データ型。
    - 浮動小数点数型
      - float
      - double
      - long double
  - void 型 ···関数値や仮引数が無いことを明示したい時などに使う。
- 派生型
  - 配列 ···同じデータ型の記憶領域の並び
  - 関数 ···例えば (char (*)(int,double)) は、...
  - ポインタ···例えば (int *) は ...
  - 構造体 ···様々の型のデータを組み合わせて 1 つのデータ型として扱う。
  - 共用体 ···1 つのデータ領域を複数のデータ型で共用する。

以下、この節では

基本的なデータ型である整数型, 浮動小数点数型

についてまとめ、

複数の (同じ型とは限らない) データを組み合わせて

1つのデータとして扱う仕組み (構造体, 共用体)、

新しく構成したデータ型に名前を付ける方法

を紹介する。

また、上の分類の示す様にC言語では文字を表すためのデータ型が特別に設けられている訳ではないので、C言語における文字や文字列の扱いについて簡単に触れておく。

11–1 整数型： char, short, int, long, unsigne

- 整数値を表すためのデータ型としては char, short, int, long (および、各々をunsignedにしたもの) が用意されているが、**ANSI規格**で定められているのは次のことだけである。

(あとはコンピュータ／処理系に依存。)

$$
\begin{aligned}
\texttt{char}\text{のビット数} &= 8\,\text{ビット} \\
\texttt{long}\text{のビット数} &\geq \texttt{int}\text{のビット数} \\
&\geq \texttt{short}\text{のビット数} \\
&\geq 16\,\text{ビット} \\
\texttt{long}\text{のビット数} &\geq 32\,\text{ビット}
\end{aligned}
$$

- 標準ヘッダファイル <limits.h> の中に、扱える最大整数値などの情報が入っている。

マクロ名		意味
CHAR_BIT	···	charのビット数
INT_MIN	···	intの最小値
INT_MAX	···	intの最大値
LONG_MIN	···	longの最小値
LONG_MAX	···	longの最大値
	·····	

- 整数値を表すための最も標準的なデータ型は int で、普通1ワードに格納される。

表せる範囲：

- 8ビット、16ビット、32ビットで表せる最大整数、最小整数は各々次の通り。

ビット数		表せる最小整数	表せる最大整数
signed	8	−128	127
	16	−32768	32767
	32	−2147483648	2147483647
unsigned	8	0	255
	16	0	65535
	32	0	4294967295

- 普通のC言語処理系だとオーバーフローのチェックはしてくれない。

  ⇨ C言語では、オーバーフローしない様にするのはプログラマの責任。

例えば：

```
[motoki@x205a]$ nl datatype-int-overflow-Kelley.c Enter
     1 #include <stdio.h>
     2 #define  BIG  2000000000

     3 int main(void)
     4 {
     5   int  a, b=BIG, c=BIG;

     6   a = b + c;
     7   printf("a=%d  b=%d  c=%d\n", a, b, c);
     8   return 0;
     9 }
[motoki@x205a]$ gcc datatype-int-overflow-Kelley.c Enter
[motoki@x205a]$ ./a.out Enter
a=-294967296  b=2000000000  c=2000000000
[motoki@x205a]$
```

整数定数：

- 普通の整数定数は int, long, またはunsigned long型 のデータとして扱われる。　(表せる最小の型が選ばれる。)

- 整数定数の型を long, unsigned, ... などに指定することができる。例えば、

  37u, 37U は unsigned 型
  37l, 37L は long型
  37ul, 37UL は unsigned long型

- 8進整数を10進整数と混同しないこと。例えば、

  456 は 10進定数、
  0456 は 8進定数 (10進で $4\times8^2+5\times8^1+6\times8^0=302$ に相当)
  0x456 は 16進定数 (10進で $4\times16^2+5\times16^1+6\times16^0=1110$ に相当
  0xaBc は 16進定数 (10進で $10\times16^2+11\times16^1+12\times16^0=2748$ に

整数の内部表現方式：

- unsignedの場合の整数データの記憶方式は全て同じ。すなわち、長さが n のビット列

$$b_{n-1}\ b_{n-2}\ b_{n-3}\cdots\cdots b_2\ b_1\ b_0$$

が符号なし整数を表すと見た場合、その値は

$$\sum_{i=0}^{n-1} b_i \times 2^i$$

である。

- signedの場合の整数データの記憶方式は、ほぼ全て同じ。すなわち、普通の計算機の場合、長さが n のビット列

$$b_{n-1}\ b_{n-2}\ b_{n-3}\cdots\cdots b_2\ b_1\ b_0$$

が符号付き整数を表すと見た場合、その値は

$$-b_{n-1} \times 2^{n-1} + \sum_{i=0}^{n-2} b_i \times 2^i$$

である。

11–2 浮動小数点数型

浮動小数点数型：

- 精度の保証された広範囲の実数値を表すためのもの。
  ⇨ 誤差に注意。

- `float`(単精度),
  `double`(倍精度, C言語では標準),
  `long double`(4倍精度)
  } の3つが用意されている。

- データ領域の大きさはコンピュータに依存している。
  `float`の精度 $\leq$ `double`の精度 $\leq$ `long double`の精度

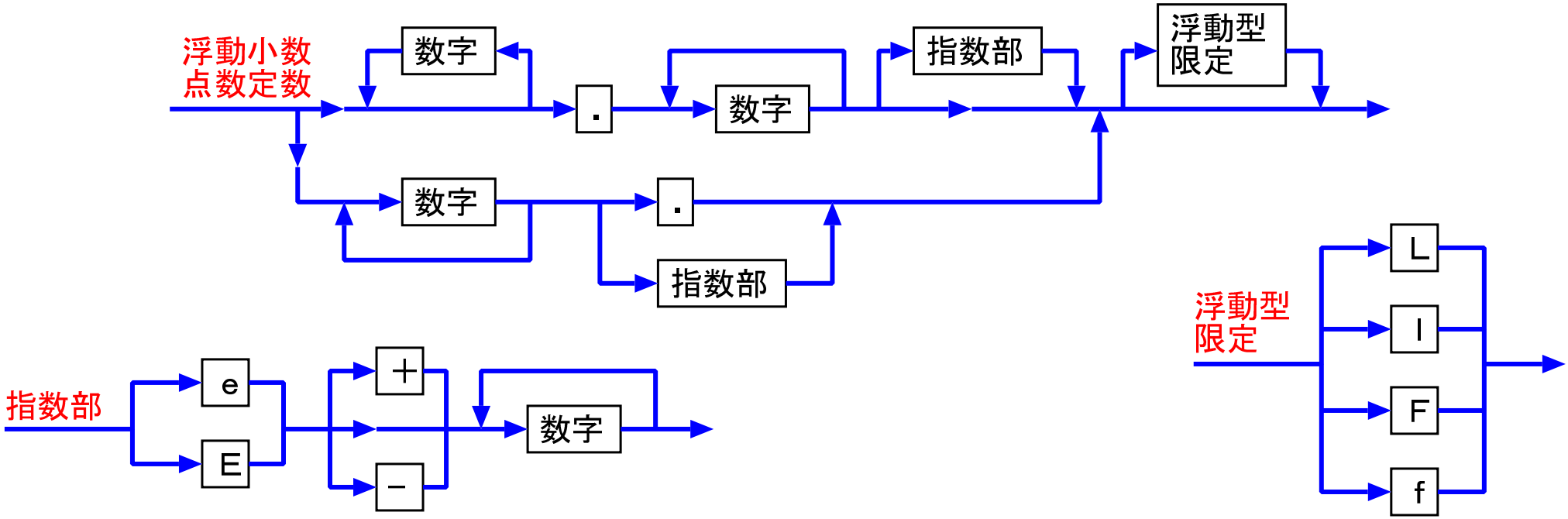
- 標準ヘッダファイル <float.h> の中に、扱える最大の浮動小数点数などの情報が入っている。訂正

マクロ名		意味
FLT_MAX	···	最大のfloat型浮動小数点数
DBL_MAX	···	最大のdouble型浮動小数点数

.....

浮動小数点数定数：

- 123.4, 123., .4, 123.4e5, .4E+5, 123e-5, ... といった書き方が出来る。 これらはdouble型の定数になる。
- 定数をfloat型にしたければ、最後に f または F という接尾語を付ける。 例えば、123.4f, .4E+5F, ... 。
- 定数をlong double型にしたければ、最後に l または L という接尾語を付ける。 例えば、123.4l, .4E+5L, ... 。



精度と指数部の表現可能な範囲 :

- ごく普通の計算機の場合、float型, double型データは、各々4バイト, 8バイトの領域を占め、10進で各々約6桁, 約15桁の精度を持つ。 また、指数部の表現可能な範囲は、およそ、各々 -38~ +38, -308~ +308 となる。 [指数部の表現可能な範囲は計算機のアーキテクチャに大きく依存する。]

IEEE規格754

- 単精度、倍精度、4倍精度における指数部、仮数部のビット数等は次の様に定められている。

	符号部	指数部	仮数部	全部で
単精度	1ビット	8ビット	23ビット(10進で6~ 7桁)	32ビット
倍精度	1ビット	11ビット	52ビット(10進で15~ 16桁)	64ビット
4倍精度	1ビット	15ビット	112ビット(10進で33~ 34桁)	128ビット

- 単精度の場合 、32ビットの列

$$\underbrace{s}_{\text{符号部}}\ \underbrace{e_7e_6\cdots e_1e_0}_{\text{指数部}}\ \underbrace{d_1d_2\cdots d_{22}d_{23}}_{\text{仮数部}}$$

によって、

$$\begin{cases} (-1)^{S}\times(1+M)\times 2^{E} & \text{if } -127<E<128 \\ (-1)^{S}\times M\times 2^{E+1} & \text{if } E=-127 \\ \text{Inf(無限大)} & \text{if } E=128,\ M=0 \\ \text{NaN(非数,Not a Number)} & \text{if } E=128,\ M\neq 0 \end{cases}$$

という実数を表す。但し、

$$M=\sum_{i=1}^{23} d_i\times 2^{-i}$$

(仮数部から $d_0=1$ というビットが省かれていると暗に仮定し、
$1.d_1d_2\cdots d_{22}d_{23}$
という2進小数を仮数部が表すと考える。)

$$E=\sum_{i=0}^{7} e_i\times 2^{i}-127$$

11–3 C言語における文字の扱い — 整数型 char

- C言語では、文字は小さな整数として扱われる。例えば a という文字を表したい時にはプログラムの中では文字定数 'a' を用いるが、これは内部では 97 という整数として扱われる。

印字可能文字の場合					
文字定数	'a'	'b'	'c'	.....	'z'
(8進表記)	'\141'	'\142'	'\143'	.....	'\172'
(16進表記)	'\x61'	'\x62'	'\x63'	.....	'\x7A'
文字の番号	97	98	99	.....	122
文字定数	'A'	'B'	'C'	.....	'Z'
文字の番号	65	66	67	.....	90
文字定数	'0'	'1'	'2'	..... '9'	
文字の番号	48	49	50	 57	
文字定数	'&'	'*'	'+'	.....	
文字の番号	38	42	43	.....	

機能文字の場合			
文字定数	'\0'(ナル文字)	'\a'(警告)	'\b'(後退)
(8進表記)	'\000'	'\007'	'\010'
(16進表記)	'\x00'	'\x07'	'\x0'
文字の番号	0	7	8
文字定数	'\t'(水平タブ)	'\n'(改行)	'\v'(垂直タブ)
文字の番号	9	10	11
文字定数	'\f'(紙送り)	'\r'(復帰)	'\"'(2重引用符)
文字の番号	12	13	34
文字定数	'\''(引用符)	'\\' (バックスラッシュ)	
文字の番号	39	92	

- 文字の番号 (小さな整数) を記憶するためのデータ型として char 型が用意されている。

- char 型は次のいずれかと同等。(コンパイラ次第)
  - signed char
  - unsigned char

- 文字定数 'a', 'b', ... は実は int 型。

- **char型変数は1バイトの領域を占める。**

  例えば 定数 'a' と同じ値をもつchar型の領域は計算機内部で次の様に表される。

7	6	5	4	3	2	1	0
0	1	1	0	0	0	0	1

  ⇨ 文字 a の番号$= 2^6 + 2^5 + 2^0 = 97$

  一般に 、ビット列

  $$b_7\ b_6\ b_5\ b_4\ b_3\ b_2\ b_1\ b_0$$

  は、unsigned char型データとしては

  $$\sum_{i=0}^{7} b_i \times 2^i$$

  という値を持ち、signed char型データとしては

  $$-b_7 \times 2^7 + \sum_{i=0}^{6} b_i \times 2^i$$

  という値を持つ。

  負数は2の補数で表す。

⇨ 表せる整数の範囲は 、

unsigned char型なら 0~ 255、

signed char型なら −128~ 127。

- C言語におけるchar型は整数型の一種に他ならないので、記憶した整数値を数字列で出力することは勿論出来る。ただ、char型変数に文字を記憶させたい場合のために、入力した文字からその番号を割り出したり、記憶した整数番号の文字を出力したり出来るようになっている。[実は、こちらの方がデータ変換が無くて単純。]

⇨ ◇ int型(またはshort型,long型)でも文字を表せる。
◇ char型変数は小さな整数値を保持するためにも使える。

例11. 6 (C言語における文字の扱い)

```
[motoki@x205a]$  nl datatype-char-Kelley.c
  1  #include <stdio.h>
  2  int main(void)
  3  {
  4    char c='a';
  5    printf("%c %c %c\n", c, c+1, c+2);
  6    printf("%d %d %d\n", c, c+1, c+2);
  7    return 0;
  7  }
[motoki@x205a]$  gcc datatype-char-Kelley.c
[motoki@x205a]$  ./a.out
a b c
97 98 99
[motoki@x205a]$
```

与えられた番号の文字を出力

整数値を10進表記で出力

11–4 C言語における文字列の扱い — char型配

C言語における文字列の表し方について：

- char型の配列を使う。
- 文字列の終わりの印として文字列の最後にヌル文字'\0'を置く。
  ⇨ (配列の大きさ) ≥ (文字列の長さ) + 1 でなければならない。

0	1	2	3	4	5	6	
's'	't'	'r'	'i'	'n'	'g'	'\0'	···

- 文字列を2重引用符で囲めば文字列定数になる。
- char型配列で文字列を表す場合は、初期設定を次の様に行うことが出来る。

```
char s[]="string";
```

(`char s[]={'s','t','r','i','n','g','\0'};` と同等。)

- 文字列操作のライブラリ関数が多数用意されている。
  ⇨ この講義ノート10.8節等を御覧下さい。

補足：

◇ 文字列定数の値はその文字列が確保されている領域へのポインタになっている。

⇨ `char *p="abc";` という宣言も出来る。

`"abc"[1]` や `*("abc"+2)` は文法的に正しい式になる。

◇ **2つの宣言の違い:**

`char *p="abc";` ···pという名前のポインタのために記憶領域が確保される。

⇨ 計算している内にpが”abc”という文字列を指さなくなることもある。

`char a[]="abc";` ···aは定数ポインタ。

注意：

◇ ヌル文字 `'\0'` を出力しないよう気を付けること。

[印字可能文字ではなく機能文字であるため。]

例題11. 8 (文字列操作のライブラリ関数)　長さが 10 以下の英単語 w と１行が80文字以下の英文章を読み込み、英文章中に現れる単語 w を全て大文字に変換して得られる文章を出力するCプログラムを作成せよ。

(考え方)

最初に英単語 w を読み込むことにすれば、あとは

① 英文章の次の１行の読み込み,
② 読み込んだ１行の中に現れる単語 w を全て大文字に変換,
③ 変換後の１行の出力

という作業を繰り返すだけである。

(考え方)

最初に英単語 w を読み込むことにすれば、あとは

① 英文章の次の1行の読み込み,
② 読み込んだ1行の中に現れる単語wを全て大文字に変換,
③ 変換後の1行の出力

という作業を繰り返すだけである。

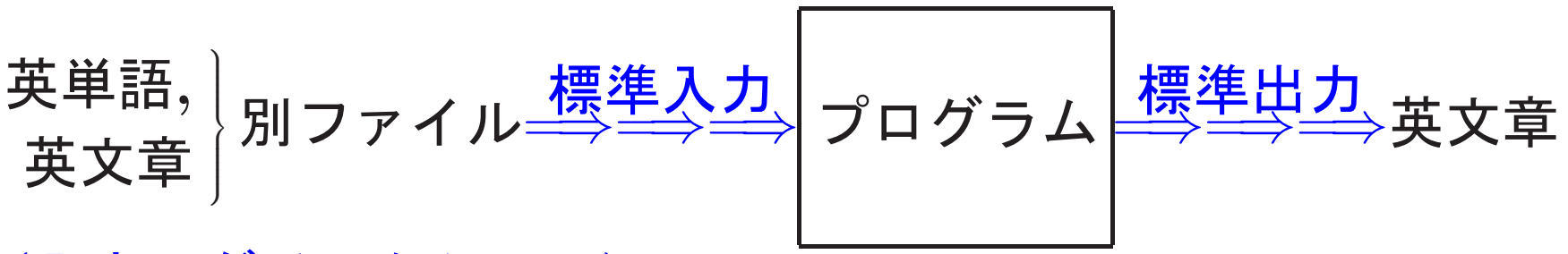
ここで、

- 英文章の次の1行を読み込むためには 、
  fgets() というライブラリ関数を利用できる。
  (⟹ 10.8節を参照)
- 文字列の中から小さな文字列パターンを探索するためには 、
  strstr() という文字列操作のライブラリ関数を利用できる。
- 英単語 w の長さを測るためには 、
  strlen() という文字列操作のライブラリ関数を利用できる。
- 英字を大文字に変換するためには 、
  toupper() という文字種類変換のライブラリ関数を利用できる。

(プログラミング)

英単語 w を保持するために　　char 型配列 word[] を、
英単語 w を大文字化したものを保持するために
　　char 型配列 WORD[] を、
1 行分の文字列を保持するために char 型配列 line[] を、
読み込んだ 1 行分の文字列を前から順に走査するために
　　ポインタ変数 remaining_seq を
用意した。

そして、次の様なデータ入力を想定して、プログラムを構成した。



(入力リダイレクション)

```
[motoki@x205a]$  nl toupper-some-words-in-sentences.c   Enter
     1 /* 長さが10以下の英単語 w と1行が80文字以下の英文章を */
     2 /* 読み込み、英文章中に現れる単語wを全て大文字に変換  */
     3 /* して得られる文章を出力するCプログラム              */
     4 /*                  (入力リダイレクションにより      */
     5 /*                   ファイルから入力することを想定する。) */

     6 #include <stdio.h>
     7 #include <string.h>
     8 #include <ctype.h>

     9 int main(void)
    10 {
    11   char word[11], WORD[11], line[82], *remaining_seq;
    12   int  word_length, i;        /* 英文章の1行は最長で    */
    13                               /* 80文字+改行コード+'\0' */

    14   scanf("%10s", word);     /* 大文字にする英単語を入力 */
```

```
15    word_length=strlen(word);
16    for (i=0; i<word_length; i++)
17      WORD[i]=toupper(word[i]);
18    WORD[word_length]='\0';

19    printf("単語 %s を大文字に換えて得られる文章:\n", word);
20    while (fgets(line, 82, stdin)!=NULL) {  /*次の1行を..
21      remaining_seq = line;
22      while ((remaining_seq=strstr(remaining_seq, word))
                                              !=NULL) {
23        for (i=0; i<word_length; i++)
24          remaining_seq[i]=WORD[i];
25        remaining_seq += word_length;
26      }
27      printf("   %s", line);
28    }
29    return 0;
30  }
[motoki@x205a]$
```

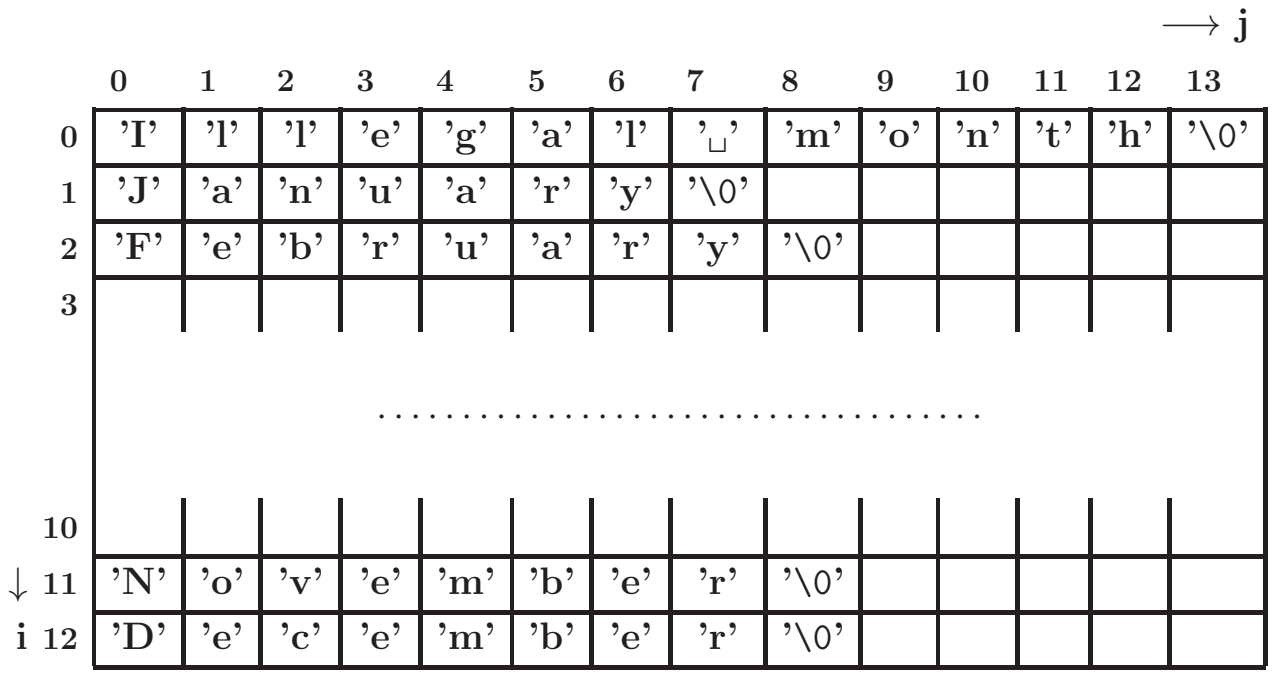
```
[motoki@x205a]$  gcc toupper-some-words-in-sentences.c  Enter
[motoki@x205a]$  cat toupper-some-words-in-sentences.data  Enter
language
------
Why C?
------
C is a small language.  And small is beautiful in programming.
C has fewer keywords than Pascal, where they are known as
reserved words, yet it is arguably the more powerful language.
C gets its power by carefully including the right control
structures and data types and allowing their uses to be nearly
unrestricted where meaningfully used.  The language is readily
learned as a consequence of its functional minimality.  C is
the native language of UNIX, and UNIX is a major interactive
operating system on workstation, servers, and mainframes.
Also, C is the standard development language for personal
computers.  Much of MS-DOS and OS/2 is written in C.  Many
windowing packages, database programs, graphics libraries, and
other large-application packages are written in C.
(A.Kelly&I.Pohl, "A Book on C" 4th ed., Addison-Wesley, 1998.)
[motoki@x205a]$  ./a.out <toupper-some-words-in-sentences.data  Enter
```

単語 language を大文字に換えて得られる文章：

```
   ------
   Why C?
   ------
   C is a small LANGUAGE.  And small is beautiful in programming.
   C has fewer keywords than Pascal, where they are known as
   reserved words, yet it is arguably the more powerful LANGUAGE.
   C gets its power by carefully including the right control
   structures and data types and allowing their uses to be nearly
   unrestricted where meaningfully used.  The LANGUAGE is readily
   learned as a consequence of its functional minimality.  C is
   the native LANGUAGE of UNIX, and UNIX is a major interactive
   operating system on workstation, servers, and mainframes.
   Also, C is the standard development LANGUAGE for personal
   computers.  Much of MS-DOS and OS/2 is written in C.  Many
   windowing packages, database programs, graphics libraries, and
   other large-application packages are written in C.
   (A.Kelly&I.Pohl, "A Book on C" 4th ed., Addison-Wesley, 1998.)
[motoki@x205a]$
```

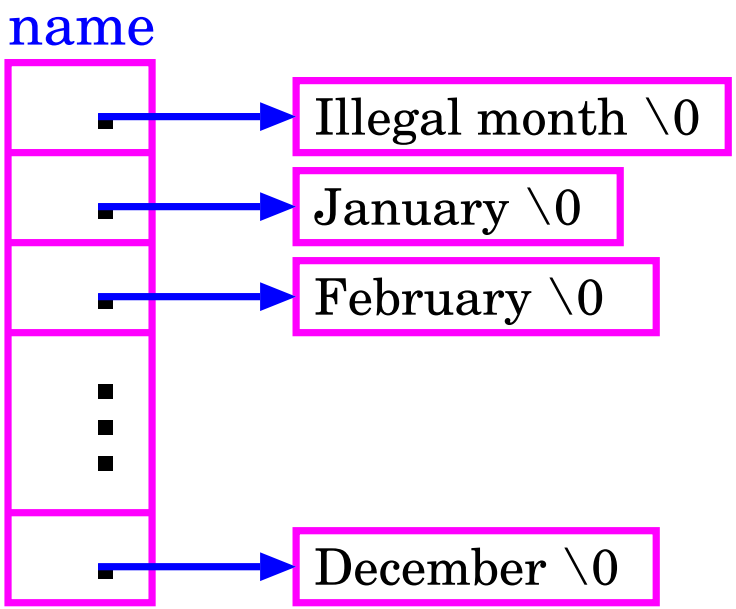
不揃い配列： 文字列自体が(1次元)配列で表されるので、複数の文字列を配列に保持しようとすると2次元のchar型配列が必要になる。

```
char name[][14]={                          /* name[k]
        "Illegal month",                   /* = k番目の月の名前 *
        "January", "February", "March",
        "April", "May", "June",
        "July", "August", "September",
        "October", "November", "December"
      };
```



しかし、この宣言を次のように書き直すとメモリが節約できる。こういった配列を不揃い配列という。

```
char *name[]={                          /* name[k]          
        "Illegal month",                /* = k番目の月の名前 *
        "January", "February", "March",
        "April", "May", "June",
        "July", "August", "September",
        "October", "November", "December"
      };
```



11–5 typedefによる新しいデータ型の定義

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. 9 (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. 10 (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。

例11. 11　宣言

```
typedef  float  Vector[10];
```

が行なわれていれば、以降では

```
Vector  v1, v2;
```

という宣言と

```
float  v1[10], v2[10];
```

という宣言は同等になる。

⇨ 上のtypedef宣言は「大きさ10のfloat型の配列」の総称として Vector 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。

11–6 構造体

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

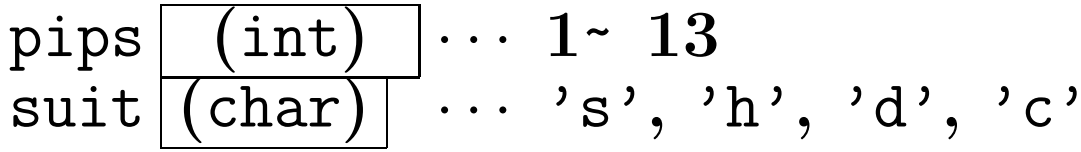
と呼ぶ。また、

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. 12 (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



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。

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. 13 (構造体要素へのアクセス; トランプのカード) 先の例11.12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. 14 (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。)

例題11. 15 (学生データの整理)　100人以下の学生について
学籍番号(5桁の英数字列), 数学の得点(100点満点)
を次々に読み込んで、各人のデータを学籍番号の順(辞書順)に出力するCプログラムを作成せよ。

(考え方)
5桁の英数字列(学籍番号)を文字列として保持するには '\0' も含めて長さが6のchar型配列があれば良く、また 0~ 100 の整数(数学の得点) を保持するには 8ビット以上の整数領域があれば良い。

⇨ 学生一人分のデータは例えば次の様な構造体で表すことが出来る。

id	(長さ6のchar配列)	···学籍番号(5桁の英数字列)を文字列として保持
math	(int型)	···数学の得点(0~ 100 の整数)を保持

学生の人数が100人以下とあるので、この構造体領域が100個連なった配列を用意すれば学生全員のデータを保持することが出来る。

例題11. 15 (学生データの整理)　100人以下の学生について
学籍番号(5桁の英数字列), 数学の得点(100点満点)
を次々に読み込んで、各人のデータを学籍番号の順(辞書順)に出力するCプログラムを作成せよ。

(考え方)　⇨ 学生一人分のデータは例えば次の様な構造体で表す。

```
   id [(長さ6のchar配列)] ···学籍番号(5桁の英数字列)を文字列として保持
math [   (int型)    ]     ···数学の得点(0~ 100 の整数)を保持
```

学生の人数が100人以下とあるので、この構造体領域が100個連なった配列を用意すれば学生全員のデータを保持することが出来る。

また、 並べ換えに関しては、
例えば例題9.6で示したバブルソート算法を用いることが出来る。

2つの文字列の大小関係(辞書順かどうか)を調べるためには、
文字列比較のライブラリ関数`strcmp()` を用いれば良い。　($\Longrightarrow$ 10.8節)

(プログラミング)

学生一人分のデータの型として Student という名前を用い、
学生全員のデータを保持するために student という名前の配列を用意した。

```
typedef struct {
    char id[6];
    int  math;
} Student;

Student student[101];
```

id (長さ6のchar配列) ···学籍番号
math (int型) ···数学の得点

エラー処理のため101人分用意

```
[motoki@x205a]$  nl sort-student-struct-data.c   Enter
     1 /* 100人以下の学生について
     2 /*     学籍番号(5桁の英数字列), 数学の得点(100点満点)
     3 /* を次々に読み込んで、各人のデータを学籍番号の順(辞書順) */
     4 /* に出力するCプログラム

     5 #include <stdio.h>
     6 #include <string.h>

     7 typedef struct {
     8     char id[6];
     9     int  math;
    10 } Student;

    11 int main(void)
    12 {
    13   Student student[101], temp;
    14   int     num, scanf_val, k, i;
```

```
15    /* データ入力 */
16    for (num=0; num<101; ) {
17      scanf_val = scanf("%5s %d",
                          student[num].id, &student[num].math);
18      if (scanf_val == 2)
19        num++;
20      else if (scanf_val == EOF)
21        break;
22      else {
23        printf("Warning: Illegal data appears.(%d-th data)\n"
                  num);
24        break;
25      }
26    }

27    if (num == 101) {
28      printf("Warning: There are 101 or more students.\n"
29             "          ==> We process first 101 students.\n");
30    }
```

データが無くなるまで101人分のデータを読み込む

データが多過ぎた場合のチェック

バブルソート

```
31    /* 辞書順に整列 */
32    for (k=0; k<num-1; k++) {
33      for (i=num-1; i>k; i--)
34        if (strcmp(student[i-1].id, student[i].id) > 0) {
35          temp          = student[i-1]; /*student[i-1]とstuden
36          student[i-1] = student[i];   /*が辞書順でないなら...
37          student[i]   = temp;
38        }
39    }

40    /* 整列後のデータを出力 */
41    printf(" id.    math.\n"
42           "-----   -----\n");
43    for (k=0; k<num; k++)
44      printf("%5s    %3d\n", student[k].id, student[k].math);
45    return 0;
46  }
[motoki@x205a]$
```

```
[motoki@x205a]$  gcc sort-student-struct-data.c  Enter
[motoki@x205a]$  cat sort-student-struct-data.data  Enter
T8100  100
T8050   78
T8022   80
T8011   50
T8064   35
T8037   90
T8001   72
T8005    0
T8046   68
T8055   46
[motoki@x205a]$  ./a.out < sort-student-struct-data.data  Ent
 id.    math.
-----   -----
T8001    72
T8005     0
```

```
T8011     50
T8022     80
T8037     90
T8046     68
T8050     78
T8055     46
T8064     35
T8100    100
[motoki@x205a]$
```

11–7 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. 18 (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。

$$\left.\begin{array}{l}\texttt{i}\\ \texttt{f}\end{array}\right\} \quad \boxed{\textbf{(int/float型)}} \quad \cdots \left\{\begin{array}{l}\text{ある時はint型のデータと見る}\\ \text{ある時はfloat型のデータと見る}\end{array}\right.$$

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 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 }
```

0x付き，8桁以上

```
[motoki@x205a]$ gcc 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。