2008 年度秋期

情報処理 IV　～ データ構造とアルゴリズム ～

火曜 3 時限 コンピュータルーム C

担当：角皆　宏・渋谷 智治

1　構造体について (復習)

構造体について簡単に復習しておこう。構造体とは、既存の型の変数の組として作られる新しい変数型 (のようなもの) で、その定義は一般的にプログラムの冒頭において行なう。構造体を構成するそれぞれの変数はメンバー (member) と呼ばれ、演算子 . を用いてその値を参照する。

実習 1.1. ベクトルの和を計算するプログラム。ベクトルの成分の個数も入力によって指定する。成分の値を保持する配列は予め大きくとってある。

```
/*   vector0.c   2008-10-07   */
#include <stdio.h>
#define MAXLEN 100

typedef struct Vector
{
    int dim;
    int val[MAXLEN];
}
Vector;

void setVector(Vector *);
void printVector(Vector);
Vector addVector(Vector, Vector);

int main( int argc, char **argv )
{
    Vector a,b,c;

    setVector(&a); printf("a="); printVector(a);
    setVector(&b); printf("b="); printVector(b);

    c=addVector(a,b);
    printf("c="); printVector(c);

    return 0;
}
```

(次頁へ続く)

今後頻繁に用いるなら、型定義や基本的な関数プロトタイプをヘッダファイルに書いておいて、
`#include "vector.h"`
などとして読込むと便利。

`Vector` 型の設計は、メンバー `dim` に成分の個数 (次元) を保持し、メンバー `val[]` (配列) の先頭から `dim` 個に各成分を順に格納することにしてみた。

(前頁より続く)

```
void setVector(Vector *v)
{
    int i;

    scanf("%d", &(*v).dim);
    for( i=0; i<(*v).dim; i++)
    {
        scanf("%d",&(*v).val[i]);
    }
}

void printVector(Vector v)
{
```

ベクトル v の内容を表示する関数 printVector() の定義の中身 (各自工夫せよ)

```
}

Vector addVector(Vector v1,Vector v2)
{
    int i;
    Vector v;

    if ( v1.dim == v2.dim )
    {
        v.dim = v1.dim;
        for( i=0; i<v.dim; i++)
            v.val[i]=v1.val[i]+v2.val[i];
    }
    else
        v.dim = 0;        /*  ERROR !! */

    return v;
}
```

setVector() は Vector 型を返す関数として設計する手もあろう。

(*v).dim
は優先順位の関係で括弧が必要 (*v.dim だと*(v.dim) と認識されてエラー)。
尚、同じ意味で
v->dim
も使える。本授業でも今後頻繁に使う予定。

&(*v).dim は優先順位の関係で、括弧がなくても &((*v).dim) の意味に取られるので、括弧不要だが、判り難ければ括弧を付けておいて差し支えない。

addVector() で引数のベクトルの次元が合わない時は、次元の値としてあり得ない値 (0) を返すようにしてみた。

addVector() は、計算結果を代入する変数をポインタで持っていって値をもらって来ることにして、返値を int 型としてエラーステータスを返すように設計する手もあろう。

実行例． 入力データを用意しておいて、入力リダイレクションを使ってみた。

```
=> cat infile
3
1 2 3
3
4 5 6
=> ./vector0 < infile
a=[ 1 2 3 ]
b=[ 4 5 6 ]
c=[ 5 7 9 ]
```

実習 1.2. 他のベクトル演算 (差・内積など) も実装してみよ。

実習 1.3. 前頁傍注にあるような関数の設計を実装してみよ。(果してどちらの使い勝手が良いだろうか。)

今の例では、成分の数 (次元) を可変 (入力による変更に対応) にするために、想定する充分な個数の要素を持つ配列を用意したが、次元が小さい場合にはかなりの無駄が生じてしまう。最近の計算機環境では比較的メモリに余裕があるとは言え、その配列を用いて多くのデータを処理することになれば、積もり積もってメモリの浪費が致命的になることもあろう。無駄をなくすには、成分の数を知ってからその分だけ変数領域を確保するようにしなければいけないのだが……。

10 年くらい前と較べると、通常のパソコンのメモリ搭載量は数十倍くらいになっている。

この例では、次元が小さければ変数 1 個につき `int` 型 100 個弱 (400byte 弱) の無駄が生じている。数百万個のデータを扱ったらどうなる？

2 メモリの動的確保

これまで、プログラムや関数においては、必要な変数や配列は全てその冒頭で宣言し、特に配列に関しては、その大きさも予め決めておく必要があった。しかし、先の例のように、実際には扱うデータ数が前もって判らなかったり、データ数が途中で変わっていくことに対応する必要があったりして、そのために予め多めに確保しておくというのでは無駄が多過ぎる、という場合もある。C 言語でもそういった場面に対応して、プログラムの実行時に必要になってから変数を作り出して使えるように、メモリを管理するための文法が用意されている。このような、実行中に新たに変数領域を確保することを**メモリの動的確保**という。

動的 ↔ dynamic
この対義語は？

実習 2.1. 最初にデータの個数を入力して、その分の配列のメモリ領域を「動的に確保」し、続いてその個数分のデータ (整数) を入力すると、その値を配列に格納した後、その総和を表示するプログラム。

ヘッダファイル stdlib.h は、関数 malloc() を使うのに必要。

```
/*   malloc1.c   2008-10-07   */
#include <stdio.h>
#include <stdlib.h>

int main( int argc, char **argv )
{
    int i, n, total = 0;
    int *p;

    printf("How many integer? ");
    scanf("%d", &n);

    p = (int *)malloc(sizeof(int) * n);

    for( i=0; i<n; i++ )
    {
        printf("[%d] = ", i);
        scanf("%d", &p[i]);
    }

    for( i=0; i<n; i++ )
        total += p[i];
    printf("total = %d\n", total);

    return 0;
}
```

考察 2.1.1. 入力したデータは何処に格納されているのか。その変数領域はプログラム中の何処で確保されたのか。

実習 2.2. 上記のプログラム中の for loop 内での、配列への添字によるアクセスは、ポインタの演算 (増加++・減少--・ポインタへの整数の加減) でも書ける。書き直してみよ。

このプログラムのポイントは次の点である。

- プログラムの実行開始時点では入力データの個数が未確定である。
- データの個数が確定してから、データを格納する為の変数領域を「**動的に**」確保している。

変数領域の動的確保には関数 `malloc()` を用いる。基本形は次の通り。　　`malloc` = Memory ALLOCation

```
 ...
int n;
type *p;
...
p = (type *)malloc(sizeof(type) * n);
...
```

(1) その型へのポインタ変数を予め宣言しておく。
(2) 関数 `malloc()` でメモリ領域を確保し、そのメモリ領域の先頭アドレスをポインタ変数に代入する。
   (a) `sizeof(type)`: 使いたい型の変数のサイズ (メモリ量) を求める。
   (b) `sizeof(type)*n`: それに必要な個数を掛けて、確保するメモリ領域量を決定。
   (c) `malloc(sizeof(type)*n)`: 所定の大きさのメモリ領域を確保して、その先頭アドレスを返値として返す。この段階では型は決まっていない (`void *`型)。
   (d) `(type *)malloc(sizeof(type)*n)`: 適切な型へのポインタに**キャスト**(明示的な型変換) する。
   (e) `p=(type *)malloc(sizeof(type)*n)`: 確保した領域の先頭アドレスを覚えておく。
(3) そのポインタ変数を用いて、確保した領域を間接参照する。

課題 1 (〆切 10/13(月)). `vector0.c` を修正して、ベクトルの成分の個数を入力してから、その分だけのメモリ領域を確保することにより、無駄なくメモリを利用するようにしたプログラム `vector1.c` を作成せよ。
(**Subject は出題日 (本課題なら** 1007**) とすること!!**)　　**extra feature 歓迎!!**