2　メモリの動的確保 (続き)

課題 1 (10/7 出題). 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。

解答例 1.1. 基本的な考え方としては、Vector 型の内部構造がどうあれ、使う時 (呼出側) では余り意識せずに使えるようにしたい。逆に言うと、呼出し方を変えずに内部構造だけ改良できるように、うまく切り分けられた設計が良い設計である。

まづは型定義と main() 関数の部分。実習 1.1 の vector0.c と、main() が全く同じで済んでいることに注意。Vector 型の定義を変更し、従って関数 setVector(), addVector() などの内容が変わっても、呼出側での使い方は全く同じで構わないように作れるのである。

```
/*   vector1.c   2008-10-07   */
#include <stdio.h>
#include <stdlib.h>

typedef struct Vector
{
    int dim;
    int *val;
}
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;
}
```

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

```
man malloc
```

で見よ。

Vector 型の定義も、例えばヘッダファイル vector.h に書いておいて、それを読込むようにしておくならば、ヘッダファイルの中で Vector 型の定義を変更するだけで、呼出す側では全く意識する必要なく、今までに書いたプログラム (ここでは関数 main()) がそのまま活用できるのである。

型の構造が決まれば、その型の変数を扱う為の関数が自然に決まってくる。

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

    scanf("%d", &v->dim);
    v->val = (int *)malloc(sizeof(int) * v->dim);
    for( i=0; i<v->dim; i++)
    {
        scanf("%d",&v->val[i]);
    }
}

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

    if ( v1.dim == v2.dim )
    {
        v.dim = v1.dim;
        v.val = (int *)malloc(sizeof(int) * v.dim);
        for( i=0; i<v.dim; i++)
        {
            v.val[i] = v1.val[i] + v2.val[i];
        }
    }
    else
    {
        v.dim = 0;
    }

    return v;
}
```

考察 2.0.1. 実は上の解答例は、とても完成品とは言い難いものである。例えば、main() 関数内で、引き続き b=addVector(a,c); としたとき、メモリ使用上どのような非効率が起こるか。

表示する為の関数 printVector() の実装は各自工夫せよ。

v->dim
は
(*v).dim
と同じ意味。今後も良く使う。便利であるので頭に留めておこう。

&v->val[i] は優先順位の関係で、括弧がなくても &((v->val)[i]) 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。

&v->val[i] はポインタ演算を用いて、v->val+i と書いても同じ。馴れると便利かも知れないが、判り易い方で構わない。コンパイルされれば同じなので、実行効率には影響無し。

b.val の指す先のメモリがどうなるか考えよ。

3　リスト (List) 構造の設計と実装

例えば「多項式」を扱いたい時にどういう型を設計すれば良いだろうか。(簡単の為に、一変数で変数名も固定 (例えば X) としよう。)「項」の列として扱うことが考えられるが、一つの多項式の項の数は可変であって、しかも計算しているうちにも項数は変わっていくので、前節のベクトルの例よりも更に柔軟性を持った設計が必要になるであろう。

「多項式簡易電卓」は本授業の目標 (課題) の例の一つである。

3–1　リスト構造とは

「値」と「その次を指すもの」との組 を基本単位として、「始め→その次→その次→ ···」と辿っていくことで一繋がりのデータを表す構造を**リンクによるリスト (linked list)** と呼ぶ。(簡単に**リスト (list)** とも言う。) この基本単位を**節点 (node)** と言う。

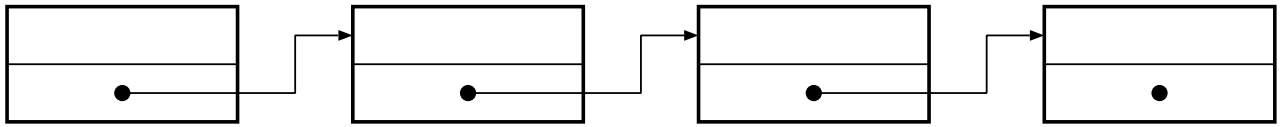

図 1: リスト構造の概念図

3–2　節点の設計

例として、int 型の値を要素として持つリストの実装を考える。節点は「値」と「その次」との組なので、構造体を用いて表すことにし、この構造体を Node 型として型名定義することにする。「その次」とは同じ型の (一般には別の) 節点であるから、この部分は Node 型へのポインタとすればよい。

```
typedef struct node {
    int val;
    struct node *next;        /* "struct" is necessary */
} Node;
```

メンバーのポインタ変数 next の宣言の段階では struct node 型を Node 型とする別名定義が終わっていないので、まだ Node 型として宣言する訳にいかず、struct node 型としている。

3–3　リストの作成・走査

実習 3.1. 整数を次々と入力し、入力した整数を各節点の値とするリストの作成。但し、0 が入力されたら終了とし、先頭からリストを辿って全ての値を順に表示する。

```
/*   list0.c   2008-10-14   */
#include <stdio.h>
#include <stdlib.h>

typedef struct node {
    int val;
    struct node *next;        /* "struct" is necessary */
} Node;

int main( int argc, char **argv )
{
    int a;
    Node head, *target;

    /* creating list */
    target = &head;
    scanf("%d",&a);
    while( a != 0 )
    {
        target->next = (Node *)malloc(sizeof(Node));
        target = target->next;
        target->val = a;
        scanf("%d",&a);
    }
    target->next = NULL;

    /* printing list */
    target = &head;
    while( target->next != NULL )
    {
        target = target->next;
        printf(" -> %d", target->val);
    }
    printf(".\n");

    return 0;
}
```

冒頭で宣言した Node 型変数 (節点) head は実は或る種のダミーであって、そのものの要素の値は使っておらず、head.next の先がリストの本体である。先頭や末尾は例外処理の発生する場所となるので、便利の為しばしばこのような工夫をすることがある。

Node へのポインタ変数 target は、リスト操作をする節点を指して覚えておく為の追跡用ポインタ。

繰返しになるが、

```
target->next
```

は、

```
(*target).next
```

のこと。後になると

```
target->next->next
```

なども現れてくるので、こうなると俄然便利。

実行例．データファイル infile からの入力例。

```
=> cat infile
12 34 567 8 0
=> ./list0 < infile
 -> 12 -> 34 -> 567 -> 8.
```

実際の計算では扱いたい (大量の) データが既にあることが普通だろうし、当座の動作確認に於いても何種類かのサンプルデータを用意しておくのは便利だろう。ファイルからの入力を行なうようにプログラムを書いても良いが、手軽にキーボードから入力したいこともあろうから、ここでは入力リダイレクションで対処することにしよう。

先頭・末尾での例外処理を避ける為に工夫された幾つかの方法がある。以下では、先頭にダミー節点を置き、末尾は NULL とする流儀に従って述べる。

リストの新規作成 (又は既存のリストの末尾への節点の追加) の要領は次の通り。

(1) 追跡用ポインタを先頭 (又は既存のリストの末尾、つまり追加する場所) の節点に向ける。

(2) リストに追加したい値がある間、
- それが入るべき新たな節点を動的に確保し、
- 追跡用ポインタが指す節点 (現在の末尾) の次へのリンクを、その新たに確保した節点に向ける。
- 追跡用ポインタを新たに確保した要素に進める。
- その要素に値を代入する。

を繰り返す。

(3) リストの末尾の節点の次へのポインタの値を NULL(何処も指さないことを表すポインタ型の値) にする。

末尾に NULL を入れて末尾判定をするのは**必須**。初期化されていないポインタなど、不当な領域を指すポインタを辿って、不当なメモリ領域にアクセスしようとすると、segmentation fault という実行時エラーになるので注意。

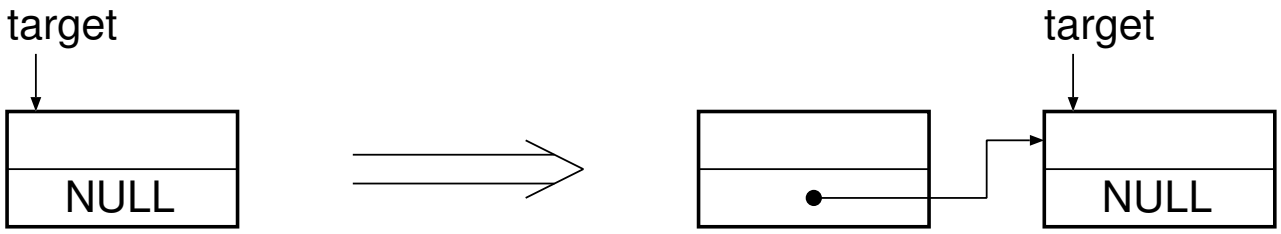

図 2: リスト末尾への節点の追加

次に、リストを辿って各節点の持つ要素を順に処理していく (この例では表示) ことを考える。これをリストの**走査** (traverse) という。リストの走査の要領は次の通り。

(1) 追跡用ポインタを先頭の節点に向ける。
(2) 追跡用ポインタの先に次の節点がある (つまり末尾でない、次が NULL でない) 間、
   - 追跡用ポインタを次の節点に進める。
   - その節点の要素を処理 (この例では表示) する。

   を繰り返す。

「リストの長さは (辿ってみないと) 予めは判らない」というのが前提なので、終了判定は回数でなく「リストが終わるまで」という形で行なう。従って、for 文より while 文の方が書き易かろう。

考察 3.1.1. 先の例で、いきなり 0 が入力された場合には、要素が 0 個のリスト (**空リスト** (empty list) という) になるが、この場合はどのような処理になるか。例外処理は必要か。

考察 3.1.2. 先頭のダミー節点を好まないなら、Node 型の変数 head を宣言してリストの先頭とせずに、Node 型へのポインタ変数として変数 head を宣言して、これをリストの先頭とする方法もある。この場合に必要な初期化処理や、例外処理を避ける実装を考えよ。

この作戦では Node 型へのポインタ型が頻出するので、こちらを基本的な型として扱うことも多い。詳細後述。

考察 3.1.3. 末尾を NULL で表す代わりに、末尾を意味するダミー節点を用意して、それに向けることで表す方法もある。この場合に必要な初期化処理や、例外処理を避ける実装は？

走査 (表示) の部分は次のようにすることも出来る。(target が一つずれた感じになっている。) 使い分けは微妙な使い勝手の差と好みとによろう。

このようにデータ構造およびそれを扱う関数の実装には色々な方法があり、一長一短な場合もある。様々な方法で実際にプログラムを書いてみて、どの方法がどんな点で有利 (便利) でどんな点で不利 (不便) か判った上で、実際のプログラミングでどれを選ぶか判断することが必要となる。その部分が、コーディング以前の真の「プログラミング」なのである。

```
/* printing list */
target = head.next;
while( target != NULL )
{
    printf(" -> %d", target->val);
    target = target->next;
}
printf(".\n");
```

課題 2 (✗切 10/20(月)). 先の例 (実習 3.1) で、リストの走査 (表示) の部分を関数 print_list() として関数化せよ。(ヒント: 関数プロトタイプは void print_list(Node *); が適当であろう。)