自分で調べるＣのポインタ

pulsar ＠物理のかぎプロジェクト
2010-03-13

Ｃ言語のポインタについて前橋さんの詳しい解説がありますが[2]，Ｃの開発環境があれば，手持ちの教科書・参考書を参照しながらテストプログラムを作ることにより，自分で疑問点を調べることができます．以下の例を変更していろいろ試みてください．本文に誤りがあったとき自信を持って訂正できます．

■ 簡単な例

まず次の例を考えましょう．結果が予想できない人は実際にプログラムを作って実行してください．

例１．次のプログラムの出力は「1, 4」．

```
#include <stdio.h>
int main(void)
{
  int n[3]={4, 5, 6}, *p=n;
  printf("%d, %d\n", p==&n[0], *p);
  return 0;
}
```

問１．例１の n, p について
(1) p[1] の値を示せ．
(2) &n[1]-&n[0] の値を示せ．

例１で実在する変数は n[0], n[1], n[2], p のみですが，「p=n;」によって n[i] の代わりに p[i] を使うことができます．一般に

```
p[i]==*(p+i), &p[i]==p+i
```

したがって p==&p[0], *p==p[0] であり，「p=n+1;」とすると

```
p[-1]==n[0], p[0]==n[1], p[1]==n[2]
```

となります．Ｃで「&p[i]==p+i」と定めたのは「p+=sizeof(int);」の代わりに「p++;」を使いたいためであると思われます．『アドレス＋整数』と『アドレスーアド

レス』は次元（？）が違うので &n[1]-n[0]== sizeof(int) と定めることもできますが(※)，&n[1]!=&n[0]+(&n[1]-&n[0]) となるので，Ｃの言語仕様はそうなっていません．実際

```
#include <stdio.h>
int main(void)
{
  int n[3], k0=&n[0], k1=&n[1];
  printf("%d, %d\n", k1-k0, &n[1]-&n[0]);
  return 0;
}
```

を実行すると，k1-k0==sizeof(int), &n[1]-&n[0]==1，2*(&n[1]-&n[0])==2 となります．「k0=&n[0];」は警告が出るかもしれませんが，無意味な代入ではないのでエラーにはなりません．

（※）Ｃの表現における &n[1]==n+1 && &n[1][2]!=(n+1)+2 等の不自然な式がなくなります．どう定めても複雑な式には歪が伴います．

■ ２次元配列とポインタの配列

プログラム例を示す前に「int n[2][2], *p[2], (*q)[][2];」で宣言される n, p, q について説明します．Ｃの記法ではありませんが

```
「int       n[2][2];」「int      *p[2];」「int       (*q)[][2];」
『int[2]    n[2];   』『int*     p[2]; 』『int[2][]   *q;       』
『int[2][2] n;      』『int*[2]  p;    』『int[2][]*  q;       』
```

と書き換えることにより，n は整数の２次元配列，p は整数へのポインタの１次元配列，q は整数の２次元配列へのポインタであることが分かりやすくなります．なお，「int m[2][3][4];」と宣言されているとき，

```
&m[i][j][k]-&m[0][0][0]==4*(3*i+j)+k
```

であることは

```
for(i=0; i<2; i++)for(j=0; j<3; j++)for(k=0; k<4; k++){
  printf("%d,", &m[i][j][k]-&m[0][0][0]);
}
```

を実行することで確認できます．また &m[i][j][k] に対して

```
m[i][j]+k==&m[i][j][k], m[i]==m[i][0], m==m[0]
```

と定められています．ただし，k が 0 でないとき m[i][0] に m[i] を代入できません．m[0] と m についても同様です．

例２．次のプログラムの出力は「5, 7」.

```
#include <stdio.h>
int main(void)
{
  int n[2][2]={{4, 5}, {6, 7}}, *p[2], (*q)[][2];
  q=p[1]=&n[0][1];
  printf("%d, %d\n", p[1][0], (*q)[1][0]);
  return 0;
}
```

問２．例２の n, p, q について
(1) ***q の値を示せ.
(2)「n[1]=p[1];」がエラーとなる理由を述べよ.

以下では簡単のため「A==B && B==C && C==D」を『 A == B == C == D 』のように略記します．例２のプログラムでは p[1]==&n[0][1] ですから,

『 p[1][0] == *p[1] == *(&n[0][1]) == n[0][1] == 5 』

です．q の方は初期値 &n[0][1] を用いて

(*q)[0][0]==*(&n[0][1]), (*q)[1][0]==*(&n[0][1]+2*1+0)

と解釈され，**(*q) == n[0][1], (*q)[1][0] == n[1][1] となります．『 q == *q == **q == &n[0][1] 』であることは分かり難いのですが,

(*q)[i][j]==q[0][i][j]

であり「int m[2][3][4];」のときの 『 m == *m == **m == &m[0][0][0], ***m == m[0][0][0] 』と類似の関係になっています．なお n[1] はアドレスであって（ポインタとは異なり）メモリ上に領域が与えられていないので，値を代入することはできません.

```
  n 『n[0][0]=4;』『n[0][1]=5;』『n[1][0]=6;』『n[1][1]=7;』
  p 『p[0]      』『p[1]=(※);』
 &q 『q=p[1];   』
    (※)==&n[0][1], (*q)[0][0]==*p[1], (*q)[0][1]==*(p[1]+1)
```

■ 文字列とポインタ

Ｃには string 型がないため，文字列は先頭をアドレス，末尾を制御文字'\0'で表わします．例えば，宣言「char s[]="ABC";」は

```
  char s[4]={'A', 'B', 'C', '\0'};
```

を略記したもので，先頭のアドレスが s で s[3]=='\0' になっています．
「char *p="ABC";」という宣言は少し分かりにくいのですが，適当な場所に "ABC" を格納する配列をつくり，その先頭アドレスを p に設定します．Ｃでは文字列の評価値は先頭アドレスなので，初期状態では p=="ABC", *p=='A' です．

例３．次のプログラムの出力は「B, DE」．

```
#include <stdio.h>
int main(void)
{
  char s[2][8]={"ABC", "DE"}, (*p)[8]=s;
  printf("%c, %s\n", p[0][1], p[1]);
  return 0;
}
```

問３．例３の s, p について
(1) (*p)[2] の値を示せ．
(2) *(*p+1) の値を示せ．

「char (*p)[8]=s;」で p に設定されるのは s です．p は『char[8]』へのポインタなので p[i]==s[i], p[i][j]==s[i][j] が成立し，p[0][1]=='B', p[1]==s[1] です．また

```
  (*p)[2]==p[0][2], *p+1==p[0]+1, *(p[0]+1)==p[0][1]
```

ですから，(*p)[2]=='C', *(*p+1)=='B' です．「char (*q)[4]=s;」と宣言された q では q[1][0]==s[0][4], q[2][0]==s[1][0] です．

■ 仮引数での宣言

Ｃでは（値呼びしかできないため）関数の引数にポインタが多用されます．ポインタ p が配列と関係なければ単に *p と表わせばいいので，ここでは直接あるいは間接に配列を指すポインタについて考えます．実引数としたい配列と型を合わせたいときは

```
  int n1[4];         -->  (int p1[])         または (int *p1)
  int n2[4][5];      -->  (int p2[][5])
  int n3[4][5][6];   -->  (int p3[][5][6])
  int *m1[4];        -->  (int *q1[])        または (int **q1)
  int *m2[4][5];     -->  (int *q2[][5])
```

のように宣言します．p1, p2, p3, q1, q2 がすべて配列ではなくポインタであることは，関数内で sizeof(p1) 等を printf() で表示することによって確認できます．

例４． 次のプログラムの出力は「B, D」.

```
#include <stdio.h>
void check(char p2[][8], char *q1[])
{
  printf("%c, %c\n", p2[0][1], *q1[1]);
}
/**/
int main(void)
{
  char s[2][8]={"ABC", "DE"}, *r[2];
  r[0]=s[0]; r[1]=s[1]; check(s, r);
  return 0;
}
```

問４． 例４の p, q について
(1) **q1 の値を示せ.
(2) q1[1][2] の値を示せ.

仮引数の宣言の [] を [N]（N は適当な定数）で置換した大域変数や局所変数の宣言を考えると，アドレスの計算に N は不要であることが分かります．「char s[2][8];」のとき &s[i][j]-s[0][0]==8*i+j ですが，仮引数の宣言「int p2[][8]」でも同様に

```
  &p2[i][j]-&p2[0][0]==8*i+j
```

です．したがって p2[0][1]==s[0][1] となります．また「char *r[2];」の r[i] は実在するポインタ，「char *q1[]」の q1[i] は計算したアドレスですが，q1[i]==*(q1+i), q[i][j]==*(q1[i]+j) なので，q1==r であれば q1[i][j]==r[i][j] が成立します．このとき *q1[1]==r[1][0], **q1==r[0][0], q1[1][2]='\0' です．　宣言として「char *q1[]」の代わりに「char **q1」を用いても同じですが，局所変数に対しては「char **q;」を「char *q[];」のように宣言することはできません.

なお，必ずしも実引数と仮引数の型を合わせるのがよいとは限りません．一例を次に示します.

```
#include <stdio.h>
extern int sum, n[10][10][10];
void add(int k, int *p)
{
  while(k>0){sum+=*p; p++; k--;}
}
/**/
int main(void)
{
  sum=0; add(1000, n); printf("sum=%d\n", sum);
  return 0;
}
```

■ 構造体メンバの参照

構造体のメンバ x.m は p==&x であるポインタを用いて p->m で参照できます．このことに詳しくない人は，まず構造体の基礎を学んでください．

例５．次のプログラムの出力は「ADC」．

```
typedef struct{char s[8], *p;} str2;
void main_5(void)
{
  str2 x[2]={{"ABC"}, {"DE"}}, *q=x+1;
  x[0].p=x[1].s; x[1].p=x[0].s; x[0].s[1]=q->s[0];
  printf("%s\n", x[0].s);
}
```

問５．例５の x, p について
(1) q->s[1] の値を示せ．
(1) q->p[1] の値を示せ．

「x[1].p=x[0].s;」の実行後に x[0].s[i] を書き換えても x[1].p[i] も変わるので，q->p[1]=='D' です．

■ 補遺

ポインタの性質を理解するときの留意点は，ポインタが参照するデータの構造はポインタの宣言のみで決まるということです．例えば「int n[2][3], *p=&n[0][1];」と宣言されていれば，『(int*) p=&n[0][1];』なので

```
p[-1]==n[0][0], *p==n[0][1], p[1]==n[0][2]
```

となります．つまり p から見れば p[i] は n に関係なく p+i にあるデータ *(p+i) に過ぎないのです．細かい補足は以下に列挙します．

1. printf() の書式の %p は処理系に依存しますが，エラーになる条件が %d や %u と変わるとは思えません(※)．例えば &n[0]+&n[1] は %u でもエラーになります．数値を調べるだけであれば「&n[1]-&n[0] は %p か %d か？」と思案する必要はないでしょう．

   (※)自分の処理系での数値だけに着目し，[3]のような難しい説明は気にしないことにしましょう．

2. 「int n[][2]={4, 5, 6};」は「int n[2][2]={{4, 5}, {6, 0}};」として扱われます．「int p[];」や「int p[][2];」をポインタの宣言に使えないのは，

このことが関係しているものと思われます．配列要素を参照するときは (*p)[i][j] より p[0][i][j] の方が分かりやすいでしょう．参照先によっては p[k][i][j] も使えます．

3. 「char *p;」は「char* p;」と同じですが「char* p, q;」は「char *p, q;」として処理されます．「char* p;」「char* q;」をまとめるには「typedef char *cptr; cptr p, q;」のようにします．
4. Ｃのバイブル[1]に複雑な宣言の例として「char (*(*x[3])())[5];」が挙げられていますが，これを

```
「char                 (*(*x[3])())[5];」
『char[5]             *(*x[3])();     』
『char[5]*            (*x[3])();      』
『()(char[5]*)        *x[3];          』
『(()(char[5]*))*     x[3];           』
『(()(char[5]*))*[3]  x;              』
```

と変形すると，x が「array [3] of pointer to function returning pointer to array [5] of char」であることが少し分かりやすくなるかも知れません（すぐれた教育用言語である Pascal での表現に近づきます）．

5. 自分でプログラムを書くときは，なるべく素直な表現を使いましょう．scanf() で変数 x[i][j] にデータを入力するとき，&x[i][j] の方が x[i]+j より素直です．ポインタの更新も独立した文で行いましょう．言語仕様からいえば「printf("%p, %d\n", p++, *p);」の結果は処理系に依存します．

■ あとがき

教科書・参考書を謙虚に学ぶことが基本ですが，疑問点を自力で解決しようとする姿勢も重要です．このことを奨励するために，比較的取り組みやすい例を示しました．

参考文献

[1] B. W. カーニハン，D. M. リッチー著，石田晴久訳，プログラミング言語Ｃ，第２版，共立出版，1989，ISBN4-320-02483-4.

[2] 前橋和弥，配列とポインタの完全制覇，http://kmaebashi.com/programmer/pointer.html

[3] LP64 データ型モデルへの変換，http://docs.sun.com/source/806-4836/conv_v9.html.