## ● 今日の実習

**【サンプルプログラム】**

★ ex13-1.c **簡単な再帰の例 (1: 線形漸化式)**.

1. 次の漸化式で定義される数列を計算する.

$$a_{n+1} = 2a_n + 1, \quad a_0 = 1.$$

2. a[n] を求める関数として, foo(n) を作り, 上の定義式通りにプログラムしてみる. (ex13-1-1.c)
3. 関数 foo の中で, 「自分自身の呼び出し」が含まれている.
   「**帰納的定義（漸化式）＝再帰的関数呼び出し**」
4. ex13-1-2.c は, 同じ計算を再帰を使わずに書いたもの.

★ ex13-2.c **簡単な再帰の例 (2: 階乗)**.

1. $a_n = n!$ は, 次の漸化式で定義された数列と思うことができる.

$$a_{n+1} = n \times a_n, \quad a_0 = 1.$$

2. a[n] を求める関数として, factorial(n) を作り, 上の定義式通りにプログラムしてみる. (ex13-2-1.c)
3. ex13-2-2.c は, 同じ計算を再帰を使わずに書いたもの.

★ ex13-3.c **再帰の方法**

1. このプログラム内では２種類の再帰が書かれている. いずれの関数も, 引数をプリントするだけの単純な関数である.
2. recursion_head では, 最初に自分自身を呼び出した後, 引数をプリントする.
3. recursion_tail では, 最初に引数をプリントし, その後に自分自身を呼び出す.
   再帰呼び出しの後に何も行わない呼び出しとなっている. これを「**末尾再帰**」(**tail recursion**) と呼ぶ.
4. 一見, 何の変哲もないプログラムのように見えるが, recursion_tail は自分自身を呼び出した後に, 何の作業も行っていないことに注意しよう.

★ ex13-4.c **高速乗算**

1. 「中間レポート」に出題した「高速乗算」を再帰で書いてみよう.
2. 「高速乗算」のキーポイントは, $a^{2n} = (a^n)^2$ を用いて巾乗を計算することであった. したがって, この部分を再帰で書くことが可能となる. これをそのまま再帰にしてみたのが ex13-4-1.c である.
3. 一方, その再帰計算に少々の工夫を行ったもの ex13-4-2.c を書くことができる.
4. 高速乗算は, 本質的には $A_0 = 1$, $A_n = (A_{n-1})^2$ によって定義された数列を求める問題とも理解できる.

**● これらのプログラムの実行時間の計測**

試しに, ex13-1, ex13-2, ex13-4 の各プログラムの実行時間の計測を行ってみよう. そのためには, 例えば ex13-1-1.c ならば foo(10) の呼び出しを一定回数（１万回など）を行って, その実行時間を計測すればよい. （プログラム例は ex13-1-time.c. 関数 “gettime” の中身は気にしないことにしよう.）





ただし, 他のプロセスの影響を可能な限り避けるため, N 回の関数呼び出しを M 回交互に行って, 合計時間を M で割った値である. (N, M は問題ごとに異る.)

実際の実行時間の計測例は以下のようになる. （単位「秒」）

| | 再帰 | 非再帰 | (ex13-4-1.c) |
|---|---|---|---|
| ex13-1 | 0.093902 | 0.028047 | |
| ex13-2 | 0.103447 | 0.030577 | |
| ex13-4 | 0.018104 | 0.015128 | 0.091896 |

ただし, ex13-4 については「再帰版」として ex13-4-2.c を用いている.

このように「再帰」で書いたプログラムは「非再帰」のものよりも（一般に）低速であることがわかる. さらに ex13-4-1.c に関しては, ex13-4-2.c と比較して, 極めて遅くなっていることがわかる.

**【ここまでのまとめ】**

- 「再帰」または「再帰的関数呼び出し」とは, **関数内部において自分自身の関数を呼び出すこと**である.
- 問題が「帰納的定義」（漸化式）で表現されているときには, 再帰を用いることができる.
- 再帰は, プログラムが問題の帰納的定義そのものを書けばよいので, アルゴリズムの実装が非常に容易になる.
- 再帰を用いるとプログラムの実行速度が低下する. （理由は次回）
  また, 再帰の用い方によっては, 速度の顕著な低下が見られる.

**【課題】**

★ exercise-13-1 ２つの正の整数を求めるユークリッドの互除法のプログラムを再帰を用いて書きなさい.

★ exercise-13-2 Fibonacci 数列

$$a_{n+2} = a_{n+1} + a_n, \quad a_0 = 0, \quad a_1 = 1$$

の第 $n$ 項を求めるプログラムを, 再帰を用いたもの, 再帰を用いないものの両方を書きなさい.
また, 可能ならばそれらのプログラムの実行時間を計測しなさい.

★ exercise-13-3 ex13-4-1.c と ex13-4-2.c のプログラムにおいて, それぞれのプログラムで関数 uint_power の呼び出し回数を求めなさい.

★ exercise-13-4 与えられた文字列を「逆順に」出力するプログラムを再帰を用いて書きなさい.
この問題は, 以前に例示した strrev を再帰を用いて実現するのではなく, 関数内部で出力を行うプログラムを書きなさいということです.

**【ヒント】** 関数に渡された文字列の長さが「１文字」になったら, その文字を出力する関数を再帰的に呼び出せばよい.

★ exercise-13-5 （**難？**）次の仕様をみたす関数をつくりなさい.

**【形式】**

```
char *strrev(char *s)
```

**【機能説明】**

文字列 s を逆転した文字列に書き換えます. 戻り値は s へのポインタです.

**【ヒント】**

s の末尾文字へのポインタを作成し, その文字を一旦退避してから, そこに NULL 文字を書き込めば, 再帰で呼び出した関数内部での文字列の長さが２減ります.

**【注意】**

この問題は K&R に「問題」として掲載されています.

★ exercise-13-6 「**ハノイの塔**」 (**Hanoi Tower**) とは以下のような問題です.

**【ハノイの塔】**

３本の柱があり, そのうちの１本の柱に大きさの異る $n$ 枚の円盤が, 下から大きな順に並んでいます. （他の２本の柱には円盤はありません.）この $n$ 枚の円盤を以下の条件の元に, 他の２本のいずれかの柱に, 下から大きな順に並ぶように移動させなさい.

1. １回に移動できる円盤の枚数は１枚のみである.
2. 小さな円盤の上に大きな円盤を載せることはできない.

☆ exercise-13-6-1 ハノイの塔を解くアルゴリズムを示し, それを証明しなさい.

☆ exercise-13-6-2 $n$ 枚の円盤に関するハノイの塔を解くために必要な移動回数を求めなさい.

☆ exercise-13-6-3 再帰を用いてハノイの塔を解く手順を出力するプログラムを書きなさい.

**【注意】** 「ハノイの塔」は次の意味で極めて有名な問題です.

1. 問題とその手順が単純であるが, その「手数」（移動回数）が非常に高速に増加する.
2. 再帰を用いると容易にプログラム可能である.

**【その他】 電子メールで「今日の講義の感想や意見」を送ってください.**

## ★ ex13-1-1.c の内容

```
/* $Id: ex13-1-1.c,v 1.3 2004-07-02 08:31:45+09 naito Exp $ */
/* a_{n+1} = 2 a_n + 1, a_0 = 1 を計算 */
#include <stdio.h>
int     foo(int) ;
int main(int argc, char **argv)
{
  printf("%d\n", foo(10)) ;
  return 0 ;
}
int foo(int n)
{
  if (n == 0) return 1 ;
  return 2*foo(n-1) + 1 ;
}
```





## ★ ex13-1-2.c の内容

```
/* $Id: ex13-1-2.c,v 1.3 2004-07-02 08:31:54+09 naito Exp $ */
/* a_{n+1} = 2 a_n + 1, a_0 = 1 を計算 */
#include <stdio.h>
int     foo(int) ;
int main(int argc, char **argv)
{
  printf("%d\n", foo(10)) ;
  return 0 ;
}
int foo(int n)
{
  int   i, k=1, m=1 ;
  for(i=0;i<n;i++) {
    m = 2*k + 1 ;
    k = m ;
  }
  return m ;
}
```

## ★ ex13-2-1.c の内容

```
/* $Id: ex13-2-1.c,v 1.3 2004-07-02 08:48:00+09 naito Exp $ */
#include <stdio.h>
int     fractional(int) ;
int main(int argc, char **argv)
{
  printf("%d\n", fractional(10)) ;
  return 0 ;
}
int fractional(int n)   /* n! を計算 */
{
  if (n == 0) return 1 ;
  return n*fractional(n-1) ;
}
```

★ ex13-2-2.c の内容

```
/* $Id: ex13-2-2.c,v 1.3 2004-07-02 08:47:59+09 naito Exp $ */
#include <stdio.h>
int     fractional(int) ;
int main(int argc, char **argv)
{
  int   i ;
  printf("%d\n", fractional(10)) ;
  return 0 ;
}
int fractional(int n)   /* n! を計算 */
{
  int   i, m=1 ;
  if (n == 0) return 1 ;
  for(i=1;i<=n;i++) m *= i ;
  return m ;
}
```

★ ex13-3.c の内容

```
/* $Id: ex13-3.c,v 1.5 2004-07-06 09:23:50+09 naito Exp $ */
#include <stdio.h>
int     recursion_head(int) ;
int     recursion_tail(int) ;
int main(int argc, char **argv)
{
  printf("ret=%d\n", recursion_head(9)) ;
  printf("ret=%d\n", recursion_tail(9))   ;
  return 0 ;
}
int recursion_tail(int n)
{
  printf("%d\t", n) ;
  if (n == 0) return 0 ;
  return recursion_tail(n-1) ;
}
int recursion_head(int n)
{
  if (n == 0) { printf("%d\t", n) ; return 0 ; }
  recursion_head(n-1) ;
  printf("%d\t", n) ;
  return n ;
}
```





★ ex13-4-1.c の内容

```
/* $Id: ex13-4-1.c,v 1.4 2004-07-02 09:55:37+09 naito Exp $ */
/* 巾乗（高速乗算） */
#include <stdio.h>
unsigned int    uint_power(unsigned int, unsigned int) ;
int main(int argc, char **argv)
{
  printf("%u\n", uint_power(2,10)) ;
  return 0 ;
}
/* 高速乗算 */
unsigned int uint_power(unsigned int a, unsigned int n)
{
  if (n == 0) return 1 ;
  if (n&1) return a*uint_power(a, n>>1)*uint_power(a, n>>1) ;
  return uint_power(a, n>>1)*uint_power(a, n>>1) ;
}
```

★ ex13-4-2.c の内容

```
/* $Id: ex13-4-2.c,v 1.3 2004-07-02 09:55:50+09 naito Exp $ */
/* 巾乗（高速乗算） */
#include <stdio.h>
unsigned int    uint_power(unsigned int, unsigned int) ;
int main(int argc, char **argv)
{
  printf("%u\n", uint_power(2,10)) ;
  return 0 ;
}
/* 高速乗算 */
unsigned int uint_power(unsigned int a, unsigned int n)
{
  unsigned int  m ;
  if (n == 0) return 1 ;
  m = uint_power(a, n>>1) ;
  if (n&1) return m*m*a ;
  return m*m ;
}
```

★ ex13-4-2.c の内容

```
/* $Id: ex13-4-2.c,v 1.3 2004-07-02 09:55:50+09 naito Exp $ */
/* 巾乗（高速乗算） */
#include <stdio.h>
unsigned int    uint_power(unsigned int, unsigned int) ;
int main(int argc, char **argv)
{
  printf("%u\n", uint_power(2,10)) ;
  return 0 ;
}
/* 高速乗算 */
unsigned int uint_power(unsigned int a, unsigned int n)
{
  unsigned int  m ;
  if (n == 0) return 1 ;
  m = uint_power(a, n>>1) ;
  if (n&1) return m*m*a ;
  return m*m ;
}
```





★ ex13-1-time.c の内容

```
/* $Id: ex13-1-time.c,v 1.1 2004-07-02 10:01:34+09 naito Exp $ */
#include <stdio.h>
#include <sys/time.h>
#define N       100000
#define M       10
#define MICRO   1000000
unsigned long gettime(void) ;
int     foo_recursive(int) ;
int     foo_non_recursive(int) ;
int main(int argc, char **argv)
{
  int   i, j ;
  unsigned long start, end, recursive=0UL, non_recursive=0UL ;
  for(j=0;j<M;j++) {
    start = gettime() ;
    for(i=0;i<N;i++) foo_recursive(10) ;
    end = gettime() ;
    recursive += end - start ;
    start = gettime() ;
    for(i=0;i<N;i++) foo_non_recursive(10) ;
    end = gettime() ;
    non_recursive += end - start ;
  }
  printf("recursive:\n") ;
  printf("\t%lu.%.06lu secs\n", (recursive/M)/MICRO, (recursive/M)%MICRO) ;
  printf("non_recursive:\n") ;
  printf("\t%lu.%06lu secs\n", (non_recursive/M)/MICRO, (non_recursive/M)%MICRO) ;
  return 0 ;
}

/* ここに foo の recursive 版 "foo_recursive"
 *           non-recursive 版 "foo_non_recursive"
 * を入れる */
unsigned long gettime(void)
{
  struct timeval        tp ;
  gettimeofday(&tp, NULL) ;
  return (unsigned long)tp.tv_sec*MICRO + tp.tv_usec ;
}
```

## ● 前回のキーポイント

- Cにおける「**ポインタ**」とは「他のオブジェクトのアドレスを値に持つ変数」のことであり, どのような型のオブジェクトを指し示すものかを込めて定義を行う.
- ポインタではないオブジェクトに対して「アドレス演算子」 `&` を作用させると, そのオブジェクトのアドレスを得ることができる. 逆に, ポインタに対して「間接演算子」 `*` を作用させると, そのポインタの指し示すオブジェクトの値を得ることができる.
- （１次元）配列の識別子は, 配列オブジェクトの先頭アドレスを示す**定数**ポインタと考えることができる.
- ポインタに対する「加算」及び「インクリメント・デクリメント」は, 「アドレスに対して `n` バイトを加算」するのではなく, そのポインタが指し示す型のオブジェクトの「`n` 個先」のオブジェクトのアドレスを得る操作である.
- 配列の要素参照 `a[i]` は `*(a+i)` と等価な操作である.
- ポインタとして定義された変数に対する「インクリメント・デクリメント」は許されているが, 配列として定義されたオブジェクトに対する「インクリメント・デクリメント」は許されない. なぜなら, 配列は「定数」ポインタであり, 定義済みの配列はメモリ内で「固定された場所」にあるからである.
- 関数引数としてポインタを利用すると, 呼び出された関数側からは, 呼び出し側のオブジェクトを参照することになり, 関数副作用として, 関数内部での値の変更を呼び出し側に反映させることができる.





## ● 前回の課題の解説

★ `exercise-12-1` 次の仕様をみたす関数をつくりなさい.

**【形式】**

```
int ext_gcd(int a, int b, int *x, int *y)
```

**【機能説明】**

２つの正の整数 `a`, `b` に対して, 拡張されたユークリッドの互除法を用いて

$$ax + by = \gcd(a, b)$$

をみたす `x`, `y` を, $xy \neq 0$ をみたすもので, 一組求めます. 戻り値は `a` と `b` の最大公約数です.

```c
/* $Id: exercise12-1.c,v 1.1 2004-06-25 15:49:54+09 naito Exp $ */
/* extended_gcd */
#include <stdio.h>

int ext_gcd(int, int, int *, int *) ;

int main(int argc, char **argv)
{
  int   a=10, b=3, x, y, gcd ;

  gcd = ext_gcd(a,b,&x,&y) ;
  printf("%d * %d + %d * %d = %d\n", a, x, b, y, gcd) ;
  return 0 ;
}

int ext_gcd(int a0, int b0, int *x, int *y)
{
  int   a[3]={1,0}, b[3]={0,1} ;
  int   i, q, t ;

  a[2] = a0 ; b[2] = b0 ;
  while(b[2]) {
    printf("%d\n", b[2]) ;
    q = a[2]/b[2] ;
    for(i=0;i<3;i++) {
      t = a[i] - q*b[i] ;
      a[i] = b[i] ; b[i] = t ;
    }
  }
  while(!a[0]*a[1]) {
    a[0] += b[0] ; a[1] += b[1] ;
  }
  *x = a[0] ; *y = b[0] ;
  return a[2] ;
}
```

★ exercise-12-4 以下の標準関数を書きなさい.

☆ exercise-12-4-1 strcat

☆ exercise-12-4-2 strncat





```c
/* $Id: exercise12-4-1.c,v 1.2 2004-07-06 08:47:13+09 naito Exp $ */
/* 文字列 s1 に s2 を append する       *
 * 戻り値は append された s1            */
char *strcat(char *s1, const char *s2)
{
  char  *save=s1 ;

  while(*++s1) ;
  while((*s1++ = *s2++)) ;
  return save ;
}

/* 文字列 s1 に s2 を最大 n 文字 append する    *
 * 戻り値は append された s1                    */
char *strncat(char *s1, const char *s2, size_t n)
{
  char  *save=s1 ;

  while(*++s1) ;
  while((*s1++ = *s2++)&&(--n)) ;
  if (*s1) *s1 = '\000' ;
  return save ;
}
```

**【注意】** この strncat は連結された s1 の末尾に NULL 文字を付け加えていますが, 標準関数ライブラリの strncat にはそのような機能がついていない場合があります.