# 情報システム工学 II(システム開発工学)

## 真のプログラマとは

*Handling bits on algorithm*

**山田泰司**

taiji@aihara.co.jp

**株式会社あいはら 研究開発チーム**

# プログラミングとは

- ▷ もの作りである、か
- ▷ システム設計である、か

もの作りの側面も重要である。しかし、個人の成果物だけではまず成立しない。よって、情報システム設計とソフトウェア開発の両面の技術が必要となる。

- ▷ よりよいモノを開発するために、喜びを見出せるか
- ▷ 既にある成果物をうまく活用できるか
- ▷ 自身の成果物をうまく再利用できるか
- ▷ いい意味でのオリジナリティを発揮できるか
- ▷ ﾒ切・納期を守れるか
- ▷ 自分が書いたコードの責任範囲を理解しているか

# コーディングとは

- まずはプログラミング言語の選択〜目的に適したプログラミング言語を
- 様々な言語それぞれの利点・欠点、管理・開発コスト、依存関係…
- そして土俵となる、プラットフォーム、そしてハードウェアの理解など

さまざまなレベルでの「設計」と切り離しては考えられない。

- スクリプト言語を理解するにも、プログラミング言語Ｃ/Ｃ＋＋から !?
- プログラミング言語Ｃ/Ｃ＋＋を理解するにも、アセンブラ・機械語から !?
- アセンブラ・機械語を理解するにも、コンピュータアーキテクチャから !?

すべてを理解している事などまずあり得ない。必要なときに、すぐさま的確に学べるために、学び方を学んでおこう。

# よいプログラミングとは

▷ 『よい名付け親になれるか』
プログラミングでは、プログラム名、ファイル名、クラス名、手続き名、変数名などあらゆる場面で名前を付ける必要がある (無名クラス、無名関数、無名変数などもあるので、すべてではない)。よいコードには至るところで、体を表す的確な、しかも冗長でない名前がついている。ちなみに、当然の事ながら、改名はいつでもできる。

▷ 『適度にコメントを残せるか』
過剰なコメントはコードを埋もれさせ、肝心なコードの可読性が極めて悪い。それだけでなく、コメントのメンテナンスが疎かになり、実体と異なったコメントがついていると余計にトラブルに見舞われることになる。意識し過ぎる必要はないが、経験を積めば、大まかな流れを指し示すべき箇所、改めて理解することが困難である事を承知している箇所、問題が残されていることを承知している箇所など、ポイントを抑えたコメントが書けるようになる。

プログラミングとは、アルゴリズムを考案している以外のほとんどの時間は、実は『命名』に費やしていると言っても過言ではない。そして、よく『命名』されたコードならば、コメントさえあまり必要でなくなる。

# よいプログラミングとは（つづき）

- **『標準規格を知っているか、常に意識しているか』**
  **プログラミング言語C/C＋＋で言えば、**ANSI C89, ISO C99, (C11), ISO C++03, C++0x(C++11) **や、システムコールなどに関わるところで言えば、**IEEE POSIX.1-1996, POSIX.1-2001, X/Open XPG3**〜、特にネットワーク関係だと** RFC **など。一次情報であるマニュアルページ等を読めばそれが書いてあるはずである。**
- **『公序良俗、他人の権利を害していないか』**
  **法令遵守は当然のこととして、一般常識、モラル、良心に従ったプログラミングを行なわないと一定のペナルティを食らうことになるだろう。そして特に、知的財産権、著作者人格権などの基本的な理解を深めることが重要である。**

**この双方とも関係するが、**Google **等の検索エンジンにやたら頼っているプログラマはまずコーディングが遅い、もしくは、間違っている。一次情報に近いところにすぐさま辿り着くことが肝要である。そして、安易にネット情報を拝借するプログラマは著作権法の見地からしても危なっかしい。**

# よいコーディングをするには

- はじめから、機械語やアセンブラが必須とまでは行かないが…
- 少なくとも、メモリ上のバイナリを必要なときにイメージできること
- 論理演算、算術演算、ビット演算は基本中の基本
- 数学は出来れば出来るほどよい
- アルゴリズムは知ってれば知っているほどよい

すると、

- アルゴリズムの記述が極めて簡潔になる事がある！
  → コードの保守や拡張が容易になり、安全・確実な処理を実現できる。
- 計算量が極めて少なくなる事がある！
  → 実現可能なことが一気に広がり、高度な処理を実現できる。

→ さらには、新たなソリューションの研究開発に繋がる。

# （例題１）ビット演算を理解しよう

**プログラム** `test00.c` **を、ソースをよく読んだり、コンパイルして実行したりして、ビット演算を理解せよ。**

```
$ make CC=gcc test00
$ ./test00
          :
```

**(注) 必ず** GNU make **コマンドを使うこと。**

# （例題１）の解説１

test00.c - **よく使うビット演算の紹介**

```
        :
#define MASK_00 (1L<<0)          // 左ビットシフト数を列挙、さまざまなフラグ
#define MASK_01 (1L<<1)
#define MASK_02 (1L<<2)
#define MASK_03 (1L<<3)
#define MASK_04 (1L<<4)
#define MASK_05 (1L<<5)
#define MASK_06 (1L<<6)
#define MASK_07 (1L<<7)
        :
  long mask;
        :
  mask = 0L;                     // 00000000 すべてのフラグをオフにする
  mask = ~0L;                    // 11111111 すべてのフラグをオンにする
  mask = MASK_00 | MASK_02 | MASK_04 | MASK_06;
                                 // 01010101 あるいくつかのフラグをオンにする
  (mask & MASK_04)               // 真       あるフラグがオンかどうか調べる
  (mask & MASK_05)               // 偽       〃
  mask |= MASK_05;               // 01110101 既にあるフラグに、あるフラグを加える
  mask &= ~MASK_04;              // 01100101 既にあるフラグに、あるフラグを除く
  mask |= MASK_03 | MASK_07;     // 11101101 既にあるフラグに、あるいくつかのフラグを加える
  mask &= ~(MASK_02 | MASK_06); // 10101001 既にあるフラグに、あるいくつかのフラグを除く
        :
```

# （例題１）の解説２

test00.c - **右シフトと符合**

```
        :
  long mask;

  mask<<24>>24;            // 算術シフト

(signed)mask<<24:       10101001000000000000000000000000
(signed)mask<<24>>24:   11111111111111111111111110101001

  (unsigned)mask<<24>>24; // 論理シフト

(unsigned)mask<<24:     10101001000000000000000000000000
(unsigned)mask<<24>>24: 00000000000000000000000010101001
        :
```

# （例題１）の解説３

test00.c - 知ってると得するビット演算

```
                    10101001
    m & (m-1):      10101000 最右 1 ビットをオフにする。2^n なら零になる
                    10000000
    m & (m-1):      00000000 〃

                    00000111
    m & (m+1):      00000000 最右 0 ビットと連続する 1 ビットを 0 にする。2^n-1 なら零になる
                    00001011
    m & (m+1):      00001000 〃

                    11001000
    m & (-m):       00001000 最右 1 ビットを取り出す

                    00110111
    ~m & (m+1):     00001000 最右 0 ビットを取り出す

                    11001000
    ~m & (m-1):     00000111 末尾まで続く 0 を 1 ビット列として取り出す

                    11001000
    m ^ (m-1):      00001111 最右 1 ビットと末尾まで続く 0 を 1 ビット列として取り出す

                    11001000
    m | (m-1):      11001111 最右 1 ビットを最右ビットまで満たす

                    00110111
    m | (m+1):      00111111 最右 0 ビットを 1 とする
  (Henry S. Warren, Jr., ``Hacker's Delight,'' Addison Wesley, 2003. より)
```

要は m+1 は繰り上がり、m-1 は繰り下がり、-m は反転と繰り上がり（２の補数でのマイナス）の応用、これを知ってると吉。

# （例題２）IPv4 アドレスの操作

インターネットアドレス (IPv4) は /usr/include/netinet/in.h:

```
typedef uint32_t in_addr_t;
struct in_addr {
  in_addr_t s_addr;
};
```

のような構造体でわかるように、32 ビット符号なし整数で管理されている。
また、アドレスやアドレス空間はそれぞれ、4 オクテット表記およびマスク長 (prefix length) で表される (CIDR: Classless Inter-Domain Routing 表記)。

```
IPv4 アドレス
        210.154.62.67                   (4 オクテット表記)

IPv4 アドレス空間
        210.154.62.65/29                (4 オクテット表記/マスク長)
        210.154.62.65/255.255.255.248   (4 オクテット表記/ネットマスク)

ネットワークアドレス〜ブロードキャストアドレス
        210.154.62.64..210.154.62.71
```

任意の IPv4 アドレス空間と IPv4 アドレスが与えられたとき、その IPv4 アドレスがその IPv4 アドレス空間に属するかを判定するプログラムを書け。

```
$ make CC=gcc test01
$ ./test01 -n 133.38.237.94/26 -a 133.38.237.117
        :
$ ./test01 -n 133.38.237.94/26 -a 133.38.237.63
        :
```

# （例題２）の解説

test01.c の主要部分の抜粋

```
        :
char net_addrs[16] = "210.154.62.64", addrs[16] = "210.154.62.67";
int net_mask_len = 29;
struct in_addr net_addr, net_mask, end_addr, addr;
        :
/* マスク長 l からネットマスク (network byte order) を作成 */
net_mask.s_addr = htonl(~0UL << (32 - net_mask_len));

1ULL<<(32 - net_mask_len) /* マスク長 l のアドレス空間のサイズ */

inet_aton(net_addrs, &net_addr); /* 指定されたネットワークアドレス文字列をバイナリ (network byte order) へ変換 */

net_addr.s_addr = net_addr.s_addr & net_mask.s_addr; /* ネットマスクからネットワークアドレスを補正 */

/* ネットマスクとネットワークアドレスからネットワーク終端アドレスを生成 */
end_addr.s_addr = net_addr.s_addr | ~net_mask.s_addr;

/* アドレス空間のサイズ=ネットワーク終端アドレス-ネットワークアドレス+1 */
(unsigned long long)ntohl(end_addr.s_addr) - ntohl(net_addr.s_addr) + 1

inet_aton(addrs, &addr); /* 指定されたアドレス文字列をバイナリ (network byte order) へ変換 */

/* 指定されたアドレスがネットワーク空間に属するか、の判定 */
// ntohl(net_addr.s_addr) <= ntohl(addr.s_addr) && ntohl(addr.s_addr) <= ntohl(end_addr.s_addr)
(addr.s_addr & net_mask.s_addr) == (net_addr.s_addr & net_mask.s_addr)
        :
```

(注) IPv4 only であること強調するために敢えて inet_aton, inet_ntoa を使っているが、レガシーシステムでも動作させたいなどの理由がない限り、inet_pton, inet_ntop を使う事。
(考察) htonl, ntohl を忘れるとリトルエンディアンのアーキテクチャ(x86 系) で間違った結果となるが、それを忘れないようにするには？

# （例題３）IPv6 アドレスの操作

インターネットアドレス (IPv6) は /usr/include/netinet/in.h(もしくは/usr/include/netinet6/in6.h):

```
struct in6_addr {
  union {
    uint8_t    __u6_addr8[16];
        :
  } __u6_addr;
};
#define s6_addr __u6_addr.__u6_addr8
```

のような構造体でわかるように、16 個の 8 ビット符号なし整数の配列で管理されている。
IPv6 では CIDR はさほど重要ではないが、IPv4 互換アドレスなどがあるので CIDR 表記によるアドレス操作は有用である。

```
IPv6 アドレス
        2001:200::8002:203:47ff:fea5:3085       (8 個の 16 ビット 16 進表記（ひとつの「::」は任意数の零 16 ビットの略))

IPv4 互換アドレス
        ::ffff:210.154.62.67                    (6 個の 16 ビット 16 進表記＋ 4 オクテット表記)

IPv4 互換アドレス空間
        ::ffff:210.154.62.65/125                (〃/マスク長)
        ::ffff:210.154.62.65/ffff:ffff:ffff:ffff:ffff:ffff:ffff:fff8    (〃/ネットマスク)

ネットワーク始点アドレス〜ネットワーク終点アドレス
        ::ffff:210.154.62.64..::ffff:210.154.62.71
```

任意の IPv6 アドレス空間と IPv6 アドレスが与えられたとき、その IPv6 アドレスがその IPv6 アドレス空間に属するかを判定するプログラムを書け。

```
$ make CC=gcc test02
$ ./test02 -n ::ffff:0.0.0.0/96
        :
```

# （例題３）の解説

test02.c の主要部分の抜粋

```
        :
    char net_addrs[64] = "::ffff:210.154.62.64", addrs[64] = "::ffff:210.154.62.67";
    int net_mask_len = 32*4-(32-29)/* = 125 */;
    struct in6_addr net_addr, net_mask, end_addr, addr;
        :
    /* マスク長 l からネットマスク (network byte order) を作成 */
    net_mask = in6_addr_lsl(in6_addr_not(in6_addr_zero()), (32*4 - net_mask_len));

    inet_pton(AF_INET6, net_addrs, &net_addr); /* 指定されたネットワークアドレス文字列をバイナリへ変換 */

    net_addr = in6_addr_and(net_addr, net_mask); /* ネットマスクからネットワークアドレスを補正 */

    /* ネットマスクとネットワークアドレスからネットワーク終端アドレスを生成 */
    end_addr = in6_addr_or(net_addr, in6_addr_not(net_mask));

    inet_pton(AF_INET6, addrs, &addr); /* 指定されたアドレス文字列をバイナリへ変換 */

    /* 指定されたアドレスがネットワーク空間に属するか、の判定 */
    // in6_addr_le(net_addr, addr) && in6_addr_le(addr, end_addr)
    in6_addr_eq(in6_addr_and(addr, net_mask), in6_addr_and(net_addr, net_mask))
        :
```

128 ビット整数を扱える標準的な型は、ない。よって、多倍長ビット演算、多倍長論理演算、多倍長算術演算をコーディングする必要がある。乗算、除算以外はそれほど難しくはない。
ここでは、マクロで、要素が演算可能な指定された配列数で多倍長演算を実現し、それを 128 ビットの struct in6_addr のための演算手続きを実装。

# （例題３）の解説（つづき１）

test02.c の主要部分の抜粋（つづき１）

```
        :
#define arr_init(n, a) do {\
  memset(&(a)[0], 0, sizeof((a)[0])*n); \
} while (0)
#define arr_not(n, a, b) do {\
  int i;\
\
  for (i=0; i<n; i++)\
    (b)[i] = ~(a)[i];\
} while (0)
        :
#define arr_lsl(n, a, s, b) do {\
  int i;\
\
  for (i=0; i<n; i++) {\
    int k, l, m;\
\
    k = i + s /(8*sizeof((a)[0]));\
    l = k + 1;\
    m = s % (8*sizeof((a)[0]));\
    (b)[i] =\
      ((0<=k && k<n) ? (a)[k]<<m : 0) |\
      ((0<=l && l<n) ? (a)[l]>>(8*sizeof((a)[0])-m) : 0);\
  }\
} while (0)
#define arr_tr(n, a, tr, b) do {\
  int i;\
\
  for (i=0; i<n; i++)\
    (b)[i] = tr((a)[i]);\
} while (0)
        :
```

(補足) CPP(C プリプロセッサ) マクロの副作用による弊害ゆえ敬遠されがちなマクロ手続きであるが、内部的に正しく使えば、再利用できる、型に依存しないコーディングも可能である。しかし、やはり可読性は良くはないので C++テンプレートの利用も検討しよう。

# （例題３）の解説（つづき２）

test02.c の主要部分の抜粋（つづき２）

```
        :
struct in6_addr in6_addr_zero(void)
{
  struct in6_addr a;

  arr_init(16, a.s6_addr);
  return a;
}
struct in6_addr in6_addr_not(struct in6_addr a)
{
  struct in6_addr b;

  arr_not(16, a.s6_addr, b.s6_addr);
  return b;
}
        :
struct in6_addr in6_addr_lsl(struct in6_addr a, unsigned int s)
{
  struct in6_addr b;

  arr_tr(4, (uint32_t *)a.s6_addr, ntohl, (uint32_t *)a.s6_addr);
  arr_lsl(4, (uint32_t *)a.s6_addr, s, (uint32_t *)b.s6_addr);
  arr_tr(4, (uint32_t *)b.s6_addr, htonl, (uint32_t *)b.s6_addr);
  return b;
}
int in6_addr_cmp(struct in6_addr a, struct in6_addr b)
{
  return memcmp(&a.s6_addr[0], &b.s6_addr[0], sizeof(struct in6_addr));
}
#define in6_addr_eq(a, b) (in6_addr_cmp(a, b) == 0)
#define in6_addr_le(a, b) (in6_addr_cmp(a, b) <= 0)
        :
```

(補足) マクロのままでは活用しにくいので、型を固定した（ここでは 128 ビットの `struct in6_addr` のメンバ `s6_addr`）の手続きを実装。
(考察) `in6_addr_lsl`(logical shift left) 以外は、`in6_addr_cmp` でさえも、`ntohl` を使っていない。ループ内でのコスト高を嫌っての `ntohl` の必要性であるが、基本的に `struct in6_addr` ではエンディアン変換は不要であるのはなぜか。

# （例題４）ビットシフト、３次元グラフィックス

v_cube3.c の立方体の頂点、線分を返す手続きを理解しよう。また、その他の可視化に係る箇所をソースコードから理解しよう。

```
        :
void cube3(double lb[3], double ub[3], double vertices[8][3], int segments[12*2])
{
  int i, j;

  for (i=0; i<3; i++)
    for (j=0; j<12/3; j++) {
      int v0 = 0, v1 = 0, k;
      for (k=0; k<3; k++) {
        v0 = ((i>k)?((j>>k)%2):(i==k)?0:((j>>(k-1))%2)) + 2*v0;
        v1 = ((i>k)?((j>>k)%2):(i==k)?1:((j>>(k-1))%2)) + 2*v1;
      }
      segments[(12/3*i+j)*2+0] = v0;
      segments[(12/3*i+j)*2+1] = v1;
    }
  for (i=0; i<1<<3; i++)
    for (j=0; j<3; j++)
      vertices[i][j] = ((i>>j)%2) ? ub[j] : lb[j];
}
        :
```

このように、ビット演算を活用すると、テーブル等を使わずにアルゴリズムを簡潔に表す事が出来る。要は規則だった「番号付け」にある。

```
$ make CC=gcc LDLIBS='-lsocket -lnsl' v_cube3
$ ./v_cube3
        :
```

(発展) このプログラムの可視化には、X Window System, Version 11(X11) の Open GL 拡張である glx を利用している。これを期に扱えるようになろう。

# （例題５）ビットシフト、ｎ次元グラフィックス

v_cube.c の超立方体の頂点、線分を返す手続きを理解しよう。また、その他の可視化に係る箇所をソースコードから理解しよう。

```
        :
void cube(int n, double lb[], double ub[], double **vertices, int *segments)
{
  long i, j, nv = 1<<n, ns = (1<<(n-1))*n;

  for (i=0; i<n; i++)
    for (j=0; j<ns/n; j++) {
      int v0 = 0, v1 = 0, k;
      for (k=0; k<n; k++) {
        v0 = ((i>k)?((j>>k)%2):(i==k)?0:((j>>(k-1))%2)) + 2*v0;
        v1 = ((i>k)?((j>>k)%2):(i==k)?1:((j>>(k-1))%2)) + 2*v1;
      }
      segments[(ns/n*i+j)*2+0] = v0;
      segments[(ns/n*i+j)*2+1] = v1;
    }
  for (i=0; i<nv; i++)
    for (j=0; j<n; j++)
      vertices[i][j] = ((i>>j)%2) ? ub[j] : lb[j];
}
        :
```

このように、ビット演算等を活用した簡潔なアルゴリズムは、より一般化したアルゴリズムへ発展させやすくなる。

```
$ make CC=gcc LDLIBS='-lsocket -lnsl' v_cube
$ ./v_cube
        :
```

（発展）`diff -u v_cube3.c v_cube.c`コマンドでコードの差分から拡張された箇所を理解できるようになろう。

# （例題６）ビット演算、フラクタル（１）

**ヒルベルト曲線と呼ばれる空間充填曲線を描け** (v_hilbert00.c)。**そして、描画中の道のりと座標の関係を２進数で印字できるようにせよ** (v_hilbert01.c)。**さらに、マウスポインタが示す座標から、その道のりを求めよ** (v_hilbert02.c)。
**ヒルベルト曲線の座標値から道のりを求める手続き：**

```
        :
unsigned long hilbert_pos2step(int n, unsigned long pos[])
{
  long long i;
  unsigned long state = 0, s = 0, p[2] = { pos[0], pos[1] }, r;

  for (i=n-1; i>=0; i--) {
    r = 4*state | 2*( (p[0] >> i) & 1 ) | ( (p[1] >> i) & 1 );
    s = (s << 2) | ( (0x361E9CB4 >> 2*r) & 3 );
    state = (0x8FE65831 >> 2*r) & 3;
  }
  return s;
}
(Henry S. Warren, Jr., ``Hacker's Delight,'' Addison Wesley, 2003. より)
        :
```

```
$ make CC=gcc LDLIBS='-lsocket -lnsl' v_hilbert00 v_hilbert01 v_hilbert02
$ ./v_hilbert00
$ ./v_hilbert01
$ ./v_hilbert02
```

**このように、ビット演算を巧妙に活用すると、下手をすると** $O(n^2)$ **の計算量となってしまうアルゴリズムが、** $O(n)$ **の計算量で済んでしまう事さえあり得る。**

# （例題７）ビット演算、フラクタル（２）

ヒルベルト曲線と呼ばれる空間充填曲線を描くプログラム v_hilbert02.c をまず拡大縮小できるようにせよ (v_hilbert03.c)。そして、指定した道のりおよびビットマスク長が指し示す範囲を描画せよ (v_hilbert.c)。
最右ビットまで続く 0 ビットの数を求める手続き、他：

```
        :
long ulntz(unsigned long x)     /* number of trailing zeros */
{
  long n = 0;

  x = ~x & (x - 1);                     // 末尾まで続く 0 を 1 ビット列として取り出す
  while (x != 0) { n++; x >>= 1; }      // 右ビットシフトで零になるまで数える
  return n;
}
        :
#define hilbert_size(order)     (1UL<<(order))                                          // 2^n
#define hilbert_length(order)   hilbert_size((order)<<1)                                // 2^(2n)
#define hilbert_max(order)      (hilbert_length(order)-1)                               // 0b111..111
#define hilbert_bitlen(order)   ((order)<<1)                                            // 2n bitsize
#define hilbert_lmask(order,l)  (hilbert_max(order) & (~0UL<<(hilbert_bitlen(order)-(l)))) // 3 -> 0b1110..
#define hilbert_maskl(order,m)  (hilbert_bitlen(order)-ulntz(m))                        // 0b1110.. -> 3
        :
```

```
$ make CC=gcc LDLIBS='-lsocket -lnsl' v_hilbert03 v_hilbert
$ ./v_hilbert03
$ ./v_hilbert -n 4
65/5
```

この問題は、まさに IP アドレスの CIDR 形式を扱うことと同等であることに注目。
（参考）このヒルベルト曲線を応用して IPv4 アドレスのある利用状況を可視化した一例が http://maps.measurement-factory.com/で紹介されている。

# 真のプログラマとは：まとめ

ビット演算の基本から応用を通して、

- IPv4 アドレスを適切に管理するコード
- IPv6 アドレスを同様に管理するために必要なコード（多倍長演算）
- アルゴリズムにおけるビット演算の効用
- 可視化との関連性と重要性（超立方体、IP アドレス空間とフラクタル）

について紹介した。今回の技術を知っているのと知っていないのとでは、プログラマとしての力量に差がつくと言っても過言ではないだろう。

また、OpenGL を X11 上で扱う glx の使用例を併せて紹介した。他には、glut というプラットフォーム間の差異を吸収する OpenGL 入門用ライブラリが存在するが、glut はあくまで入門用・サンプルコード記述用である。それを理解して使う分には問題ないが、それ以上の発展性がないのであまりお勧めできない。