12. ビット演算

(1) 確認用プログラムの準備

ビット演算とは，データを１ビットずつ処理する演算ですが，これまで出てきた加算・減算などの演算にくらべ，なじみのない方が多いかもしれません。しかし，ひとつの命令をフィールドに分けたり，特定のビットを判定するには，必要不可欠な演算です。

当然，システム記述用言語としては，これらの操作ができなくてはいけません。しかし，なじみがない方のほうが多いと思います。そのような方は，演算結果をビットごとに確認するプログラムを先に作りましょう。

このプログラムはビット構造体や共用体の例でもありますので，ぜひ，例題として保存しておくことをお勧めします。

[ビット演算確認用プログラム]

```
#include "stdafx.h"
#include "string.h"
/*(1)ビットフィールド構造体*/
struct bitdata
{ unsigned b7:1; unsigned b6:1;
  unsigned b5:1; unsigned b4:1;
  unsigned b3:1; unsigned b2:1;
  unsigned b1:1; unsigned b0:1;
};
/*(2)演算子テーブル構造体*/
struct opetab
{ char name[4];int num;
};
/*(3)演算子テーブル定義*/
static struct opetab tab[7]
={ {"" ,0},{">>",2},{"<<",2}, {"&",2},
   {"|",2},{"^",2},{"~",1}};
/*(4)文字とビットフィールド領域共有*/
union uxd{ char ch; struct bitdata bt;};
void print_bit(char ch) /*１ビット表示  */
{  if(ch==0) printf("0");
   else printf("1");
}
/* ビット列表示 */
void print_bitlist(struct bitdata ch)
{ print_bit(ch.b0);print_bit(ch.b1);
  print_bit(ch.b2);print_bit(ch.b3);
  print_bit(ch.b4);print_bit(ch.b5);
  print_bit(ch.b6);print_bit(ch.b7);
}
/* 演算子テーブル探索*/
int table_search
         (int num,struct opetab tab[])
{ int i=num-1;
  While (strcmp(tab[i].name,
                tab[0].name)!=0) i--;
  return i;
}
/* 演算用データ入力*/
void data_in(int num,int dt[])
{ int i;union uxd r;
  for(i=0;i<num;i++)
  { printf("¥n16 進 2 桁入力(%d)=",i+1);
    scanf("%x",&dt[i]);
    printf(" 演算用データ:");
    r.ch=dt[i];print_bitlist(r.bt);
} }
void ope_in(char ch[])/* 演算子入力*/
{ scanf("¥n 演算子入力:" ", %s",ch); }
void print_result(struct bitdata r)
{ printf("¥n¥n 結 果=");
  print_bitlist(r); printf("¥n");
}
int exec_op(int id,int dt[])/*実行*/
{ switch(id)
  {   case 1  : return(dt[0] >> dt[1]);
      case 2  : return(dt[0] << dt[1]);
      case 3  : return(dt[0] & dt[1]);
      case 4  : return(dt[0] | dt[1]);
      case 5  : return(dt[0] ^ dt[1]);
      case 6  : return( ~ dt[0]);
      default: return 0;
}  }
/* メイン*/
int main(int argc, char* argv[])
{ int id; int dt[2];union uxd r;
  ope_in(tab[0].name);
  while(strcmp(tab[0].name,"end") !=0)
  { id=table_search(7,tab);
    if(id != 0)
    { data_in(tab[id].num,dt);
      r.ch=exec_op(id,dt);
      print_result(r.bt);
    }
    else printf("**演算子の誤り¥n");
    ope_in(tab[0].name);
} }
```

（2）基本的なビット演算

１ビットのデータ A, B に対する基本的な演算を下表に示します。

A	B	A & B	A | B	A ^ B	~A
0	0	0	0	0	1
0	1	0	1	1	1
1	0	0	1	1	0
1	1	1	1	1	0

これらの演算を使って，あるビットがオンかオフかの判定，ビット反転，ビットごとの組合せなどの処理を行います。

（3）ビット単位の左シフト

ビットを左にシフトします。

```
    0000 0011   (10 進数 3)
         ↓
（左に１ビットシフト）
         ↓
    0000 0110   (10 進数 6)
```

［記述法］　A<< n

左シフトは，2^n 倍の乗算の替わりにも使うことができます。

（4）ビット単位の右シフト

ビットを右にシフトします。

```
    0000 00110   (10 進数 6)
         ↓
（左に１ビットシフト）
         ↓
    0000 00011   (10 進数 3)
```

［記述法］　A >> n

右シフトは，2^n 整数による除算の替わりにも使うことができます。なお，符号ビットの取り扱いが，処理系によっては，次のように異なる場合があります。

- unsigned の場合，左端の空いた部分に 0 が詰められる。（論理シフト）

- signed の場合，最も左端のビットがコピーされる。（算術シフト）

［算術シフト］
0000 0110 ⇒（右にシフト）⇒　0000 0011
1100 0110 ⇒（右にシフト）⇒　1110 0011

［論理シフト］
0000 0110 ⇒（右にシフト）⇒　0000 0011
1100 0110 ⇒（右にシフト）⇒　0110 0011

しかし，逆にいうと上記のように処理されない処理系もありますので，算術シフトを使いたい場合は，以下のように記述することをお勧めします。

```
( A >> 1) | (A & 0x80)
```

（5）ビット列表示の改良

準備したプログラムのビット列表示は，ビットフィールド構造体を使わなくても，シフトを使えば以下のように短く表現できます。

```
  void print_bitlist_dash(char ch_in)
{  char ch;int i; ch = ch_in;
  for(i = 0;i < 8; i++, ch <<= 1)
      printf((ch & 0x80)? "1":"0");
}
```

（6）演習

上記（5）で示したプログラムを用いて，確認用のプログラムを短く書き直しなさい。