

# bit 演算 + まとめ

3年 情報科学科 大島 裕樹

今回はオマケみたいなものです。最後にまとめの問題も用意しました。

**bit 演算**とは、ビット単位でデータの操作をおこなうもの。  
対象は整数に限られます。

これまで講習してきた内容に比べると、意識して使用する機会は少ないかも・・・  
が、マイコン制御などではお目にかかる機会もあるため、知らないと後々困りますので。

以下、主な bit 演算とその使用例。

## [1] & (AND)

両方のビットが 1 のとき、結果が 1 となります。 具体的には・・・

```
0 & 0 → 0
0 & 1 → 0
1 & 0 → 0
1 & 1 → 1
```

よく if 分とかで、”&”と”&&”を書き間違えるとうまく動作しないのは、”&”だと bit 演算を行うことを意味してしまう為だったりします。

では、例題でも。テキストにそのまま書き込んでみてください。

### 例題 1

```
(1) 01010101 & 11110000 =
(2) 01010101 & 00001111 =
(3) 01010101 & 10101010 =
(4) 01010101 & 01010101 =
```

## [2] | (OR)

いずれかのビットが1のとき、結果が1となります。

```
0 | 0 → 0
0 | 1 → 1
1 | 0 → 1
1 | 1 → 1
```

演算記号自体は、if文とかでもおなじみですね。  
今回も例題を用意しました。

### 例題2

```
(1) 01010101 | 11110000 =
(2) 01010101 | 00001111 =
(3) 01010101 | 10101010 =
(4) 01010101 | 01010101 =
```

## [3] ^ (XOR)

両方のビットが異なるとき、結果が1となります。

```
0 ^ 0 → 0
0 ^ 1 → 1
1 ^ 0 → 1
1 ^ 1 → 0
```

AND や OR に比べてややこしい感じがしますが、特定のビットを反転するのに使えたりします。  
ではまた例題を。

### 例題3

```
(1) 01010101 ^ 11110000 =
(2) 01010101 ^ 00001111 =
(3) 01010101 ^ 10101010 =
(4) 01010101 ^ 01010101 =
```

## [4] ~ (NOT)

ビットの反転を行います。簡単ですね。

```
~0 → 1
~1 → 0
```

XOR と違って、すべてのビットを反転します。

### 例題 4

```
(1)  ~01010101   =
(2)  ~11110000   =
```

## [5] << (左シフト)

ビットを左にずらします。書き方は  $x \ll n$  という感じで、 $n$  ビットだけ左にずれます。

右側の空いたビットには 0 が入り、左側のビットは捨てられます。

以下具体例。

```
01010101 << 1 → 10101010
01010101 << 2 → 01010100
```

ビット単位の演算なので、正の整数なら 1 ビット左シフトすると 2 倍、2 ビット左シフトすると 4 倍・・・という具合に、 $2^n$  倍に増えていきます。

## [6] >> (右シフト)

ビットを右にずらします。ずらす方向以外は左シフトと同じです。

以下具体例。

```
01010101 >> 1 → 00101010
01010101 >> 2 → 00010101
```

正の整数なら 1 ビット右シフトすると  $1/2$  倍、2 ビット右シフトすると  $1/4$  倍・・・

という具合に、 $1/2^n$  倍に減っていきます。

おそらくライトレーサの製作時など、お世話になる演算子なので、ぜひ覚えておきましょう。

## 例題 5

- (1) 01010101 << 4 → 10101010
- (2) 01010101 >> 4 → 01010100

## [ Ex ] マクロ

bit 演算ではありませんが、非常に重要なことなので、ここで説明します。

**#define** を使用すると、プログラム中に記述された特定の文字列をすべて置き換えることができます。

言葉で説明しても、良く解らるのでサンプルコードです。

### sumple.c

```
#include <stdio.h>
#define NINZU 6          //マクロで、NINZU を 6 に置き換え
int main(void)
{
    int i;
    int goukei = 0; // 合計点
    int ten[NINZU] = { 86, 67, 46, 96, 54, 72 }; // 点数
    for (i = 0; i < NINZU; i++)
    {
        goukei = goukei + ten[i];
    }
    printf("学生数 = %d\n", NINZU);
    printf("合計 = %d\n", goukei);
    return 0;
}
```

sumple.c を見てもらうと、「NINZU」を「6」に置き換えて、配列の要素数や for 文の繰り返し回数を指定しています。このように表記することで、

- ・定数の「6」を用いるよりも、「NINZU」とマクロ定義した方が、わかりやすくなる。
- ・プログラムの変更が生じ、学生数が「6」から他の数値になったときに、**#define** 定義を修正するだけで対応できる。

といったメリットがあります。

マクロを使用する際には、他の変数などと区別をするために、すべて大文字にするのが習わしです。

(強制じゃないけど)

## ◇まとめの問題

正直、つまらん内容だった今回。bit 演算なんてそんなもんです。

せっかくなので前期の復習問題でも解いていってくださいな。

条件・配列・関数あたりまで理解していれば、結構いろんなモノが作れるんじゃないでしょうか。

- 1 2つの数を入力して、和・差・積・商を求めるプログラムを作成しましょう。  
ただし、0で除算を行おうとしたときには警告メッセージを表示するようにしてください。
- 2 10個の数を配列に入れて、その中から最大のもつと最小のもの、10個の数の平均を表示するプログラムを作成しましょう。  
プログラムには必ず繰り返し文を用いること。
- 3 入力した10進数を2進数に変換するプログラムを作成しましょう。  
for文、%演算子あたりを活用すれば上手くいくのでは。  
170 → 10101010、255 → 11111111のような感じになっていればok。
- 4 簡単なゲーム作りでもしてみましようか。  
2次元配列を使用して、昔なつかしの○×ゲームを作ってみましよう。  
○×ゲームを知らない現代っ子はそこら辺の部員にきいてくださいな。  
関数とかをうまく使うと、プログラムソースもスッキリします。

## ◇オマケ

「bit 演算はわかったけど、結局何に使うの？」

そんな疑問を持つ方も多いはず。

実はbit 演算は「**フラグ管理**」に対して威力を発揮するのです。

「フラグ」っていうのは、みなさんがよく使っている「死亡フラグ」とか「負けフラグ」とほとんど同じ意味です。

## 典型的な死亡フラグ

もう何も怖くない

マミ  
v \*(ノハ)  
ξ > °ワノ  
c) 巴つ  
く/±lj  
しノ

黄色に  
死亡フラグが立つ

マミる

c) 巴つ  
く/±lj  
しノ

もうちょっと具体的に説明すると、

「プログラム内の特定の状況を、0と1で管理するもの」

といったところでしょうか？

ゲームプログラミング的に例を挙げると、

「ジャンプしたら ”ジャンプフラグ” を立てる」

「パワーアップアイテムをとったら、 ”パワーアップフラグ” を立てる」

「特定のイベントを終えたら、 ”トゥルーエンドフラグ” を立てる」

などなど・・・。もちろんゲームプログラミング以外でも使います。

肝心の使い方ですが、フラグが立っている状態を1、それ以外を0として、フラグを立てるときはOR演算を使用します。

$0000 \mid 0001 \rightarrow 0001$

これで一番下のビットが1になりました。もちろん複数のビットを同時に1にすることも可です。フラグをもとに戻すときは、AND演算とNOT演算を使用します。

$1111 \& \sim 0001 \rightarrow 1110$   
(※ $\sim 0001 \rightarrow 1110$ )

これで一番下のビットが0になりました。

このように、bit演算を使用してフラグ管理を行う場合、

1つの変数でたくさんのフラグを扱うことができます。

int型は32ビットの型なので、32個のフラグを1つの変数で管理することができます。

もちろん必要なフラグの分だけ変数を用意することもできますが、1つの変数にまとめたほうがプログラムとしてはスッキリしますよね。

以上、オマケでした。