

ビット処理

ゲームソフト分野
1年 C++

フラグ管理

ゲームの中で状態を管理するとき**フラグデータ**を使用する

例えば、それぞれの状態を管理する**変数を宣言**して、
変数を**1**か**0**かで状態を管理する方法がある

(例) キャラクターの状態

```
int  IsPoison = 0;    // 毒状態
int  IsSleep  = 0;    // 眠り状態
int  IsAtkUp  = 0;    // 攻撃力アップ
```

フラグ管理

```
//何かしらの条件
{
    //毒状態
    IsPoison = 1;
}
//何かしらの条件
{
    //眠り状態
    IsSleep = 1;
}
```

```
//何かしらの条件
{
    //攻撃力Up状態
    IsAtkUp = 1;
}
```



それぞれの状態変化を
1つずつ個別の変数で
管理している

フラグ管理

- ・状態を増やす際には**変数も増やしていく**必要がある...
使用する**メモリ量も増大**(変数につき4バイト必要)

```
int IsDefUp = 0;           //防御力アップ  
int IsBurn = 0;           //火傷状態  
...
```

- ・複数の状態を付与する場合
それぞれのフラグを変更しなければならない...

```
//毒状態になり攻撃力が下がる攻撃を受けた  
IsPoison = 1;  
IsAtkDown = 1;
```

フラグ管理

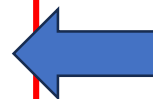
- ▶ **ビット演算**を使うと
1つの変数ですべての状態を表すことができる！

ステータスを管理する変数を1つにまとめる

unsigned int myState;

符号なし整数(マイナス値がない)
0 ~ 4294967295(2^{32})

(参考)通常のintは
-2147483647 ~ 2147483647



int IsPoison
int IsSleep
int IsAtkUp
.....

フラグ管理

～考え方～

myStateは32ビット(桁)の数値

そのうちの1ビットにひとつの状態を当てはめる

```
unsigned int myState
```

```
0000 0000 0000 0000  
0000 0000 0000 0000
```

0	0	0	0	0	0	0	0
防御力 Down	攻撃力 Down	防御力 Up	攻撃 Up	混乱	やけど	眠り	毒

フラグ管理

～考え方～

(眠り&やけど&防御力Upの状態)

```
unsigned int myState; // 0010 0110
```

0	0	1	0	0	1	1	0
防御力 Down	攻撃力 Down	防御力 Up	攻撃 Up	混乱	やけど	眠り	毒

▶状態を変化させる場合は対応するビットを1にする

フラグ管理

複数の状態を変数でセットする場合...

IsAtkDown = 1;

IsBurn = 1;

IsDefUp = 1;

IsPoison = 1;

IsConfu = 1; 変数に値を代入する処理が状態毎に必要

ビットで管理する場合は、一度に複数の状態を変更できる！

myState |= 0b01101101

0	1	1	0	1	1	0	1
防御力 Down	攻撃力 Down	防御力 Up	攻撃 Up	混乱	やけど	眠り	毒

フラグ管理

～必要な処理～

① ビットを立てる (例, 毒の攻撃を受けたので毒のビットを**1**にする)

0	0	1	0	0	1	1	1
防御力 Down	攻撃力 Down	防御力 Up	攻撃 Up	混乱	やけど	眠り	毒

② ビットを落とす (例, 毒消しが使われたので毒のビットを**0**にする)

0	0	1	0	0	1	1	0
防御力 Down	攻撃力 Down	防御力 Up	攻撃 Up	混乱	やけど	眠り	毒

③ 特定のビットが立っているか確認

(例, 攻撃力Upしてたら～, 防御力Downしてたら～)

ビット演算

OR演算 |

どちらかが1なら1

$$\begin{array}{r} 0110 \\ + 1100 \\ \hline 1110 \end{array}$$

AND演算 &

どちらも1なら1

$$\begin{array}{r} 0110 \\ \times 1100 \\ \hline 0100 \end{array}$$

XOR演算 ^

どちらも同じなら0
違う値なら1

$$\begin{array}{r} 0110 \\ 1100 \\ \hline 1010 \end{array}$$

反転 ~

0なら1,1なら0

$$\begin{array}{r} 0110 \\ \hline 1001 \end{array}$$

今回は、OR演算、AND演算、反転のみで
フラグを管理する

状態を数値で表現した例

- 通常状態 Base = 0b00000000
- 毒 Poison = 0b00000001
- 眠り Sleep = 0b00000010
- 麻痺 Para = 0b00000100
- 火傷 Burn = 0b00001000
- 攻撃↑ AtkUp = 0b00010000
- 攻撃↓ AtkDown = 0b00100000

これらの値はプログラム中で書き換えられないようにしたい...

列挙型enum

- 関連する定数をグループ化して、管理しやすくしたもの

- 文法

```
enum タグ名 { 定数1, 定数2, 定数3, ... };
```

- 例

```
enum Week {  
    Sun, Mon, Tue, Wed, Thu, Fri, Sat  
};
```

のように定義すると、自動的に

Sun:0 Mon:1 Tue:2 Wed:3 ... Sat:6
という連番の整数値になる

列挙型enum

- 番号の振り直しも可能

```
enum Week {  
    Sun, Mon, Tue = 10, Wed, Thu, Fri = 20, Sat  
};
```

とすると、

Sun:0	Mon:1	
Tue:10	Wed:11	Thu:12
Fri:20	Sat:21	

というように値を再設定することも可能で、以降は連番となる

- 定数であるため、プログラム中は書き換えられない！

列挙型BitStateの定義

```
enum BitState
```

```
{
```

```
    Base      = 0,           //000000000 通常状態
```

```
    Poison    = 1 << 0,      //000000001 どく状態
```

```
    Sleep     = 1 << 1,      //000000010 ねむり状態
```

```
    Para      = 1 << 2,      //000000100 まひ状態
```

```
    Burn      = 1 << 3,      //000001000 やけど状態
```

```
    AtkUp     = 1 << 4,      //000100000 攻撃力アップ状態
```

```
    AtkDown   = 1 << 5      //001000000 攻撃力ダウン状態
```

```
};
```

状態を付加する(ビットを立てる)

- 通常状態 Base = 0, 0000 0000
- 毒 Poison = $1 \ll 0$, 0000 0001
- 眠り Sleep = $1 \ll 1$, 0000 0010

- 通常状態を毒状態にする場合

Status | Poison

↓
Poison

0000 | 0000 (Status: Base)
0000 | 0001 (Poison)

0000 0001 (Status: Poison)

- さらに眠り状態にする場合

Status | Sleep

↓
Poison + Sleep


0000 | 0001 (Status: Poison)
0000 | 0010 (Sleep)

↓
0000 0011 (Poison / Sleep)

ステータスに各状態をOR演算することで状態を変化させる

状態を解除する(ビットを落とす)

- 通常状態 Base = 0000 0000
- 毒 Poison = 0000 0001
- 解毒 ~Poison = 1111 1110

- 毒 + ねむり状態から毒を解除にする場合
Poison & ~Poison
Sleep
0000 0011 (Poison / Sleep)
1111 1110 (~Poison)

0000 0010 (Sleep)

各状態の否定をAND演算することで状態を変化させる

状態を確認する

- ??状態が**毒**状態かどうか確認する場合

Status & **Poison**

Poison

0000 1011 (Status: ???)
0000 & 000**1** (**Poison**)

0000 000**1** (Status: **Poison**)
???状態の中に**毒**状態が含まれる

- ??状態が**眠り**状態かどうか確認する場合

Status & **Sleep**

Sleep

0000 1011 (Status: ???)
0000 & 00**1**0 (**Sleep**)

0000 00**1**0 (Status: **Sleep**)
???状態の中に**眠り**状態が含まれる

ステータスに各状態を**AND**演算することで状態を確認できる

状態を確認する

- ??状態が**麻痺**状態かどうか確認する場合

Status & **Para**



Base

0000 1011 (Status: ???)

0000 & 0**1**00 (**Para**)



0000 0000 (Status: Base)

???状態の中に**麻痺**状態は含まれていない

ステータスに各状態を**AND**演算することで状態を確認できる

フラグ操作プログラムの作成

- ビットのON/OFFを用いたフラグ操作によって
 キャラクターのステータスを変更するプログラムを
 作成する
- コマンドプロンプトから以下のコマンドを入力

```
copy nul bit01.c
```

※notepadを用いたファイル作成でもよい

列挙型BitStateの定義例

```
enum BitState
```

```
{
```

```
    Base      = 0,           //000000000 通常状態
```

```
    Poison    = 1 << 0,      //000000001 どく状態
```

```
    Sleep     = 1 << 1,      //000000010 ねむり状態
```

```
    Para      = 1 << 2,      //000000100 まひ状態
```

```
    Burn      = 1 << 3,      //000001000 やけど状態
```

```
    AtkUp     = 1 << 4,      //000100000 攻撃力アップ状態
```

```
    AtkDown   = 1 << 5,      //001000000 攻撃力ダウン状態
```

```
};
```

フラグ操作関数

- `typedef unsigned int UINT;`
`UINT myStatus = Base;`

- 状態を表示する関数(値渡し)
`void dispStatus(UINT s);`

関数内でフラグとなるビットをチェックして、どの状態にあるのかを表示する

演習 bit02.c

- bit01.cのプログラムを変更して
 - changeStatus関数で状態フラグをセット
 - clearStatus関数で状態フラグをクリア
- するプログラムを作成する

演習 bit02.c

実行例

プレイヤーの状態: 毒

1:状態セット 2:状態解除> 1

セットする状態を選択

1:毒 2:眠り 4:麻痺 8:火傷 16:攻撃↑ 32:攻撃↓> 8

プレイヤーの状態: 毒 火傷

1:状態セット 2:状態解除> 2

解除する状態を選択

1:毒 2:眠り 4:麻痺 8:火傷 16:攻撃↑ 32:攻撃↓> 1

プレイヤーの状態: 火傷

演習 bit02.c

- 実行例のように、無限ループとして以下のようにする
 - ① キャラの状態を表示
 - ② 状態の設定か解除の選択
 - ③ 状態設定ならビットを立て、解除ならビットを落とす
 - ④ ①へもどる

演習 bit02.c

- ヒント

- changeStatus関数で状態の設定

引数としてキャラの状態(myState)を受け取るが、関数内で値を変更するため、`○○渡し`で引数を受けとる

状態の設定はOR演算で行う

演習 bit02.c

- ヒント

- clearStatus関数で状態の解除

引数としてキャラの状態(myState)を受け取るが、関数内で値を変更するため、`○○渡し`で引数を受けとる

状態の設定は各状態の反転ビットとのAND演算で行う