

# システムプログラム 第4回

---

創域理工学部 情報計算科学科

松澤 智史

# 本日の内容

- 情報の解釈(前回)

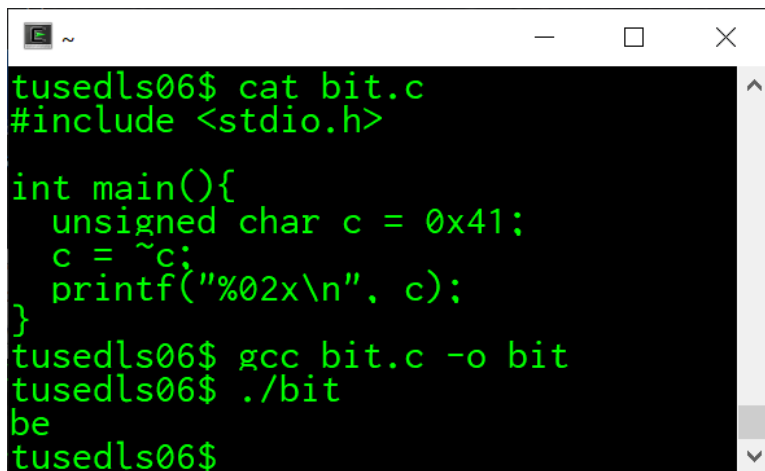
- コンピュータは情報をビットにエンコードし、ビットはバイト列として構成される
- 異なるエンコーディング(バイト列の解釈の方法)が、数値、文字列、画像・・・等に用いられる

- 情報の操作(今回)

- ビット列(ビットパタン)の操作
- 情報がビットパタンで表現されていることを知っている前提で行える操作
- 高性能なプログラムを実現するために必要なテクニック
  - 算術演算で行うよりも高速に動作するケースが多い

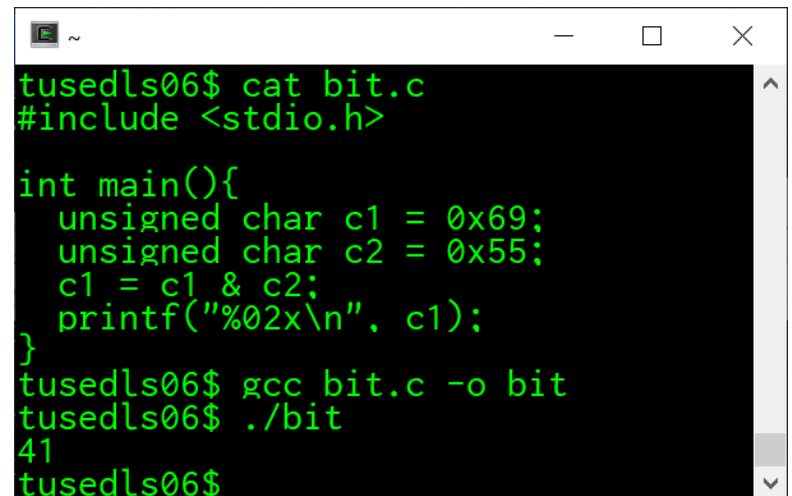
# ビットレベル演算

- NOT ~ (反転)
  - AND &
  - OR |
  - XOR ^
- 
- $\sim 0x41 \rightarrow \sim[0100\ 0001] \rightarrow [1011\ 1110] \rightarrow 0xBE$
  - $0x69 \& 0x55 \rightarrow [0110\ 10001] \& [0101\ 0101] \rightarrow [0100\ 0001] \rightarrow 0x41$



```
tusedls06$ cat bit.c
#include <stdio.h>

int main(){
    unsigned char c = 0x41;
    c = ~c;
    printf("%02x\n", c);
}
tusedls06$ gcc bit.c -o bit
tusedls06$ ./bit
be
tusedls06$
```



```
tusedls06$ cat bit.c
#include <stdio.h>

int main(){
    unsigned char c1 = 0x69;
    unsigned char c2 = 0x55;
    c1 = c1 & c2;
    printf("%02x\n", c1);
}
tusedls06$ gcc bit.c -o bit
tusedls06$ ./bit
41
tusedls06$
```

# 例: 学生の属性

1バイト(8bit)に属性情報を持たせる(解釈の仕方の決定)

- 学年(2bit) 00(B3 00, B4 01, M1 10, M2 11)
- 性別(1bit) 0(男性0, 女性1)
- 住所(1bit) 1(自宅0, 下宿1)
- 所属研究室(4bit) 1010
  - 明石研 0000 滝本研 1000
  - 安藤研 0001 桂田研 1001
  - 宮本研 0010 松澤研 1010
  - 石井研 0011 大村研 1011
  - 入山研 0100
  - 佐藤研 0101
  - 田畑研 0110
  - 野口研 0111

[00011010] 0x1A  
学生の属性情報

# 使い道の例: OR

- 論理和である「OR」は特定のビットを立てる時に利用

0	0	0	1	1	0	1	0
---	---	---	---	---	---	---	---

 0x1A

- 性別を女性1に変更

0	0	1	0	0	0	0	0
---	---	---	---	---	---	---	---

 0x20 性別ビットのみ1(女性)

OR演算

0	0	0	1	1	0	1	0
---	---	---	---	---	---	---	---

 0x1A

0	0	1	1	1	0	1	0
---	---	---	---	---	---	---	---

 0x3A

立てたいビットのみ1にしておけばOR演算の結果は元に関係なく1になる

# 使い道の例: AND

- 論理積であるAND演算子はビットを落とす時に利用

0	0	0	1	1	0	1	0
---	---	---	---	---	---	---	---

 0x1A

- 住所を自宅0に変更

1	1	1	0	1	1	1	1
---	---	---	---	---	---	---	---

 0xEF 住所ビットのみ0

AND演算

0	0	0	1	1	0	1	0
---	---	---	---	---	---	---	---

 0x1A

0	0	0	0	1	0	1	0
---	---	---	---	---	---	---	---

 0x0A

落としたいビットのみ0にしておけばAND演算の結果は元に関係なく0になる

# 使い道の例: AND(マスク処理)

- AND演算子はマスク処理を行うと特定のビットの値を抽出

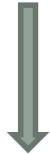
0 0 0 1 1 0 1 0    0x1A

- 所属研究室のみ取り出す

0 0 0 0 1 1 1 1    0x0F    取り出すビット1

AND演算

0 0 0 1 1 0 1 0    0x1A



0 0 0 0 1 0 1 0    0x0A

比較

0 0 0 0 1 0 1 0

松澤研究室    0x0A

```
tusedls16$ cat masksample.c
#include <stdio.h>

int main(){
    unsigned char att = 0x1A;

    if((att & 0x20) == 0x20){
        printf("女性 ");
    }else{
        printf("男性 ");
    }

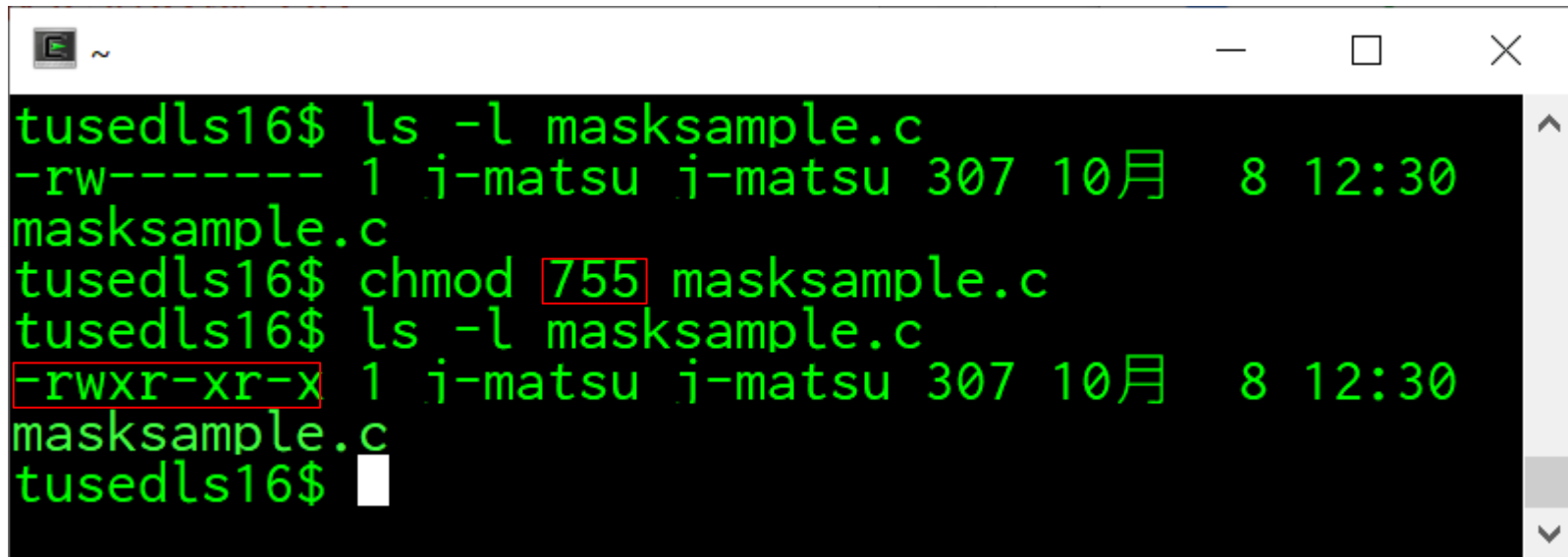
    if((att & 0x10) == 0x10){
        printf("下宿 ");
    }else{
        printf("自宅 ");
    }

    if((att & 0x0F) == 0x0A){
        printf("松澤研究室\n");
    }
}

tusedls16$ gcc masksample.c -o masksample
tusedls16$ ./masksample
男性 下宿 松澤研究室
tusedls16$
```

# 参考:UNIXのファイルパーミッション

UNIXのファイルパーミッション r(読み), w(書き), x(実行)もフラグとしてビット列で属性情報を持つ

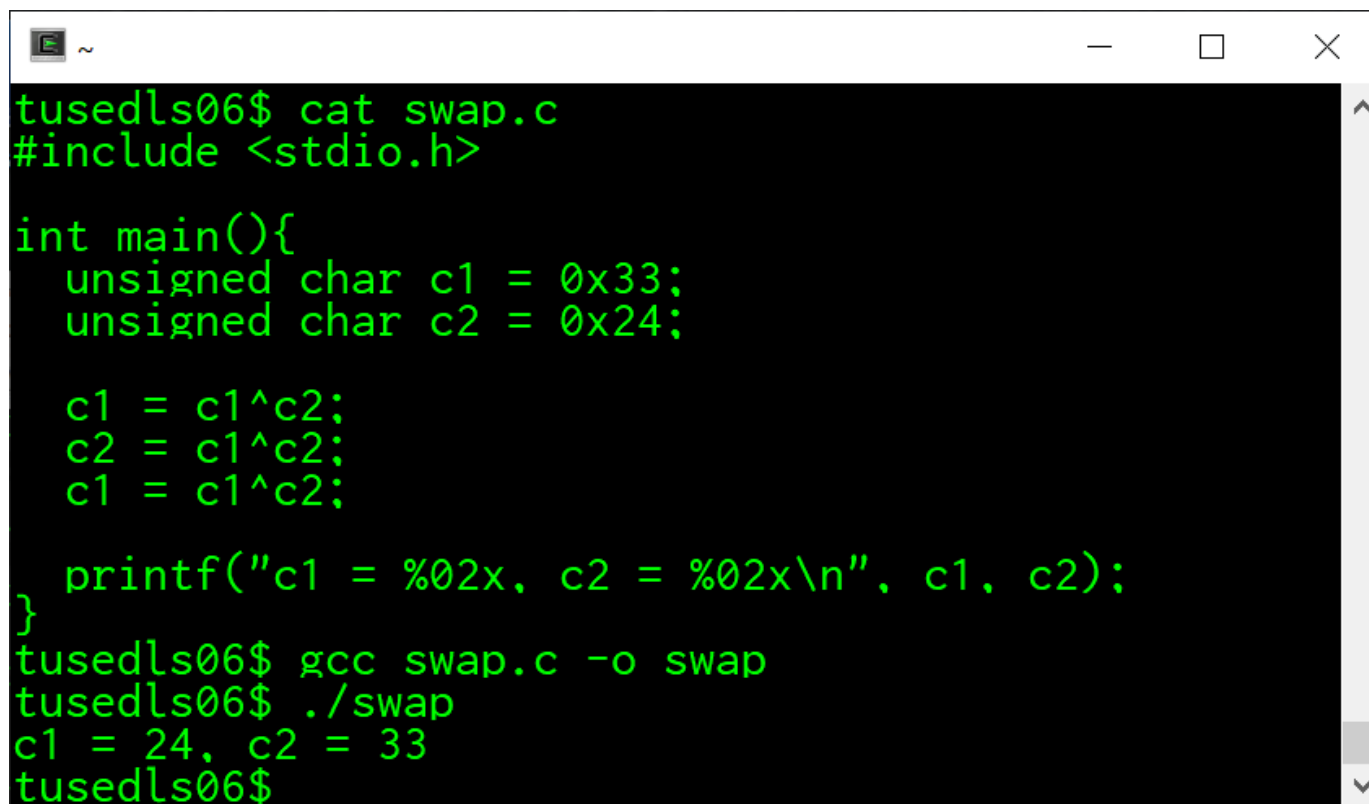


```
tusedls16$ ls -l masksample.c
-rw----- 1 j-matsu j-matsu 307 10月  8 12:30
masksample.c
tusedls16$ chmod 755 masksample.c
tusedls16$ ls -l masksample.c
-rwxr-xr-x 1 j-matsu j-matsu 307 10月  8 12:30
masksample.c
tusedls16$
```



# 応用:swap(値交換) XOR

- ビット列a,bにおいて  $a \oplus a = 0$ ,  $b \oplus a \oplus a = b$ である



```
tusedls06$ cat swap.c
#include <stdio.h>

int main(){
    unsigned char c1 = 0x33;
    unsigned char c2 = 0x24;

    c1 = c1^c2;
    c2 = c1^c2;
    c1 = c1^c2;

    printf("c1 = %02x, c2 = %02x\n", c1, c2);
}
tusedls06$ gcc swap.c -o swap
tusedls06$ ./swap
c1 = 24, c2 = 33
tusedls06$
```

参考: 1行で書くと  $c1 \oplus c2 \oplus c1 \oplus c2$ ;

$a \oplus a = 0$  を利用して変数の初期化をする書き方もある

# シフト演算

- ビットパターンを左右にずらす演算

- 左シフト

- $[00011000] \ll 1 \rightarrow [00110000]$

- 右シフト

- $[00011000] \gg 1 \rightarrow [00001100]$

- 論理シフト

- $[11000001] \ll 2 \rightarrow [00000100]$

- $[11000001] \gg 2 \rightarrow [00110000]$

- 算術シフト

- $[11000001] \ll 2 \rightarrow [10000100]$

- $[11000001] \gg 2 \rightarrow [11110000]$

```
tusedls09$ cat shift1.c
#include <stdio.h>

int main(){
    unsigned char c = 0xc1; //11000001
    unsigned char c1 = c<<2;
    unsigned char c2 = c>>2;

    printf("c1 = %02x\n", c1);
    printf("c2 = %02x\n", c2);
}
tusedls09$ gcc shift1.c -o shift1
tusedls09$ ./shift1
c1 = 04
c2 = 30
tusedls09$
```

```
tusedls09$ cat shift2.c
#include <stdio.h>

int main(){
    int i = -1; //11111111.....111
    int i1 = i<<2;
    int i2 = i>>2;

    printf("i1 = 0x%x, %d\n", i1,i1);
    printf("i2 = 0x%x, %d\n", i2,i2);
}
tusedls09$ gcc shift2.c -o shift2
tusedls09$ ./shift2
i1 = 0xfffffff, -4
i2 = 0xfffffff, -1
tusedls09$
```

# シフト演算

- 左に1回シフト→ 数値としては 2 倍
- 右に1回シフト→ 数値としては  $\frac{1}{2}$  倍

```
tusedls09$ cat shift3.c
#include <stdio.h>

int main(){
    int i = 0x1000;

    printf("0x1000 >>4 0x%08x\n", i>>4);
    printf("0x1000 <<4 0x%08x\n", i<<4);
}
tusedls09$ gcc shift3.c -o shift3
tusedls09$ ./shift3
0x1000 >>4 0x00000100
0x1000 <<4 0x00010000
tusedls09$
```

```
tusedls09$ cat shift3.c
#include <stdio.h>

int main(){
    int i = 16000;

    printf("1000 >>4 %d\n", i>>4);
    printf("1000 <<4 %d\n", i<<4);
}
tusedls09$ gcc shift3.c -o shift3
tusedls09$ ./shift3
1000 >>4 1000
1000 <<4 256000
tusedls09$
```

1/16倍, 16倍

# 乗算・除算

$$\begin{array}{r} \phantom{\times} 111 (= 7) \\ \times \phantom{111} 11 (= 3) \\ \hline \phantom{\times} \overset{1}{1} \overset{1}{1} \overset{1}{1} \\ \phantom{\times} 111 \\ \hline 10101 (= 21) \end{array}$$

乗算は最初の数进行シフト演算でずらしながら足していく

$$\begin{array}{r} \phantom{100} 11 \\ 100 \overline{) 1110} \\ \underline{100} \phantom{0} \\ 110 \\ \underline{100} \phantom{0} \\ 10 \phantom{0} \end{array}$$

割り算は右シフトによる引き算の積み重ね

# 乗算の例

- $a \times 15$  より  $a \times 16$ の方が高速
- $00010011(19) \times 00010000(16) \rightarrow$  左シフト4で終わり
- $00010011(19) \times 00001111(15) \rightarrow$   
左シフト3+左シフト2+左シフト1 + 19  
または 左シフト4 + (-19)
- 乗算で任意の値を選べる際は $2^n$ の値が好まれる

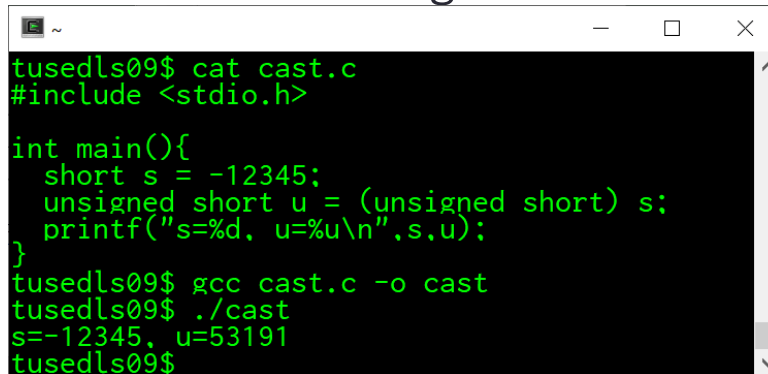
# 余談(RSA暗号)

- シフト演算とは無関係ではあるが、  
計算時間の短縮の例としてRSAの暗号化・復号処理がある
- $n = p \cdot q$  ( $p, q$ は素数),  $m$ を平文とする
- 暗号文  $c = m^e \pmod{n}$
- 平文  $m = c^d \pmod{n}$
- $e$ は適当な数,  $d = e^{-1} \pmod{(p-1) \cdot (q-1)}$
- $e$ は  $65537 = 2^{16} + 1$  が良く用いられる
- 計算量的には暗号化も復号も同じであるが、  
実際は暗号化処理の方が高速

# 符号あり・なしの変換

- C言語では異なる数値データ型をキャストすることが可能

short → unsigned short

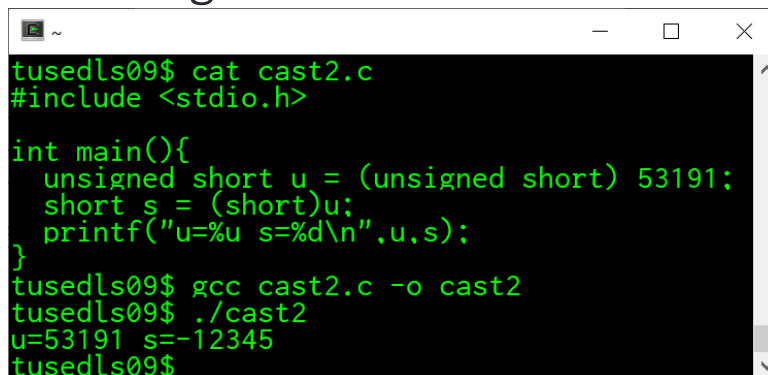


```
tusedls09$ cat cast.c
#include <stdio.h>

int main(){
    short s = -12345;
    unsigned short u = (unsigned short) s;
    printf("s=%d, u=%u\n",s,u);
}
tusedls09$ gcc cast.c -o cast
tusedls09$ ./cast
s=-12345, u=53191
tusedls09$
```

ビット列は変化せず解釈が変わる

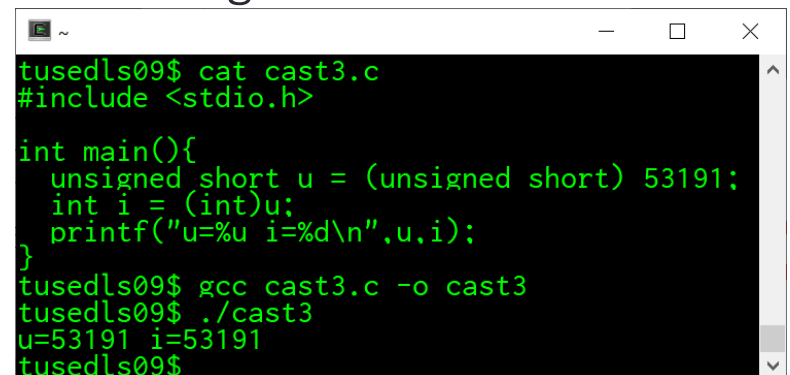
unsigned short → short



```
tusedls09$ cat cast2.c
#include <stdio.h>

int main(){
    unsigned short u = (unsigned short) 53191;
    short s = (short)u;
    printf("u=%u s=%d\n",u,s);
}
tusedls09$ gcc cast2.c -o cast2
tusedls09$ ./cast2
u=53191 s=-12345
tusedls09$
```

unsigned short → int

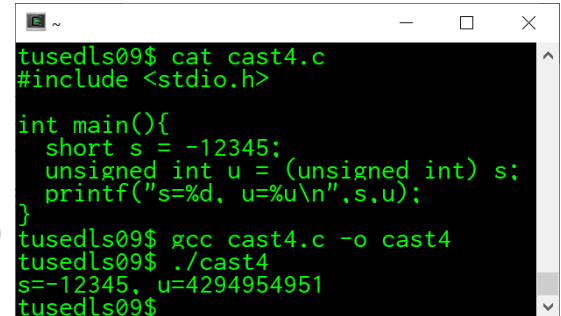


```
tusedls09$ cat cast3.c
#include <stdio.h>

int main(){
    unsigned short u = (unsigned short) 53191;
    int i = (int)u;
    printf("u=%u i=%d\n",u,i);
}
tusedls09$ gcc cast3.c -o cast3
tusedls09$ ./cast3
u=53191 i=53191
tusedls09$
```

# 符号あり・なしの変換

- 符号付き整数型A  $\Rightarrow$  符号付き整数型B
  - Aの型サイズ  $\leq$  Bの型サイズ : 値は不変
  - Aの型サイズ  $>$  Bの型サイズ : 表現可能であれば値は不変 表現不可能であれば処理系依存
- 符号無し整数型A  $\Rightarrow$  符号無し整数型B
  - Aの型サイズ  $\leq$  Bの型サイズ : 値は不変
  - Aの型サイズ  $>$  Bの型サイズ :  $A \% (B\text{の型の表現しうる最大値}+1)$
- 符号付き整数型A  $\Rightarrow$  符号無し整数型B
  - $A \geq 0$ 
    - Aの型サイズ  $\leq$  Bの型サイズ : 値は不変
    - Aの型サイズ  $>$  Bの型サイズ :  $A \% (B\text{の型の表現しうる最大値}+1)$
  - $A < 0$ 
    - Aの型サイズ  $\leq$  Bの型サイズ :  $A + (B\text{の型の表現しうる最大値}+1)$
    - Aの型サイズ  $>$  Bの型サイズ :  $-(-A \% (B\text{の型の表現しうる最大値}+1)) + (B\text{の型の表現しうる最大値}+1)$
- 符号無し整数型A  $\Rightarrow$  符号付き整数型B
  - Aの型サイズ  $<$  Bの型サイズ : 値は不変
  - Aの型サイズ  $\geq$  Bの型サイズ : 表現可能であれば値は不変 表現不可能であれば処理系依存



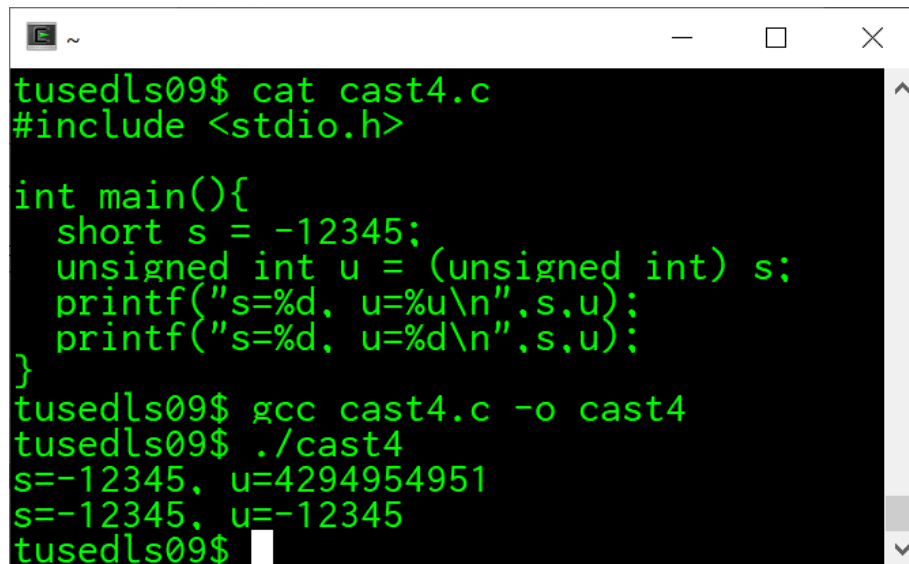
```
tusedls09$ cat cast4.c
#include <stdio.h>

int main(){
    short s = -12345;
    unsigned int u = (unsigned int) s;
    printf("s=%d, u=%u\n", s, u);
}
tusedls09$ gcc cast4.c -o cast4
tusedls09$ ./cast4
s=-12345, u=4294954951
tusedls09$
```



# 符号あり・なしの変換:補足

- 符号ありで負の数の場合で, 桁数(ビット数)を拡張する場合
  - →拡張ビットをすべて1にする
  - (char) -1 [11111111] → (short) -1[1111111111111111]



```
tusedls09$ cat cast4.c
#include <stdio.h>

int main(){
    short s = -12345;
    unsigned int u = (unsigned int) s;
    printf("s=%d, u=%u\n", s, u);
    printf("s=%d, u=%d\n", s, u);
}

tusedls09$ gcc cast4.c -o cast4
tusedls09$ ./cast4
s=-12345, u=4294954951
s=-12345, u=-12345
tusedls09$
```

# 符号あり・なしの変換：注意点

- 符号あり・なしの変換は直感的でない結果を生む場合がある
- バグの原因となりうる
  - Java等の言語では符号なしをサポートしていない

```
tusedls09$ cat castbug.c
#include <stdio.h>

int count(int a[], unsigned int length){
    int i;
    int result = 0;
    for(i=0; i<=length-1; i++){
        result += a[i];
    }
    return result;
}

int main(){
    int a[]={1,2,3};
    printf("%d\n", count(a,0));
}

tusedls09$ gcc castbug.c -o castbug
tusedls09$ ./castbug
Segmentation fault
tusedls09$
```

```
tusedls09$ cat castbug2.c
#include <stdio.h>

int count(int a[], unsigned int length){
    length--;
    int result = 0;
    while(length>=0){
        result += a[length];
        length--;
    }
    return result;
}

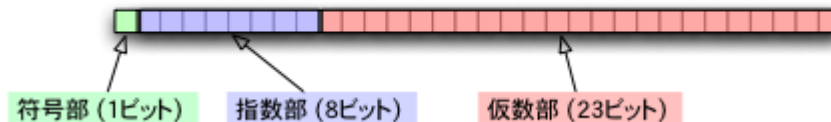
int main(){
    int a[]={1,2,3};
    printf("%d\n", count(a,3));
}

tusedls09$ gcc castbug2.c -o castbug2
tusedls09$ ./castbug2
Segmentation fault
tusedls09$
```

# 復習：浮動小数点エンコーディング

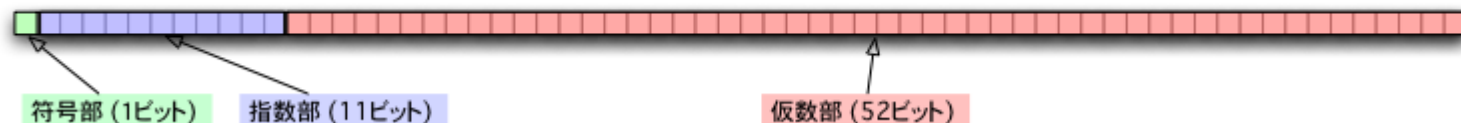
- 符号部, 指数部, 仮数部からなるエンコーディング
- float型
  - 4バイト(32ビット)
  - float の表す値 =  $(-1)^{\text{符号部}} \times 2^{(\text{指数部}-127)} \times 1.\text{仮数部}$

float 型 (4バイト=32ビット) の内部表現



- double型
  - 8バイト(64ビット)
  - double の表す値 =  $(-1)^{\text{符号部}} \times 2^{(\text{指数部}-1023)} \times 1.\text{仮数部}$

double 型 (8バイト=64ビット) の内部表現



# 浮動小数点の誤差

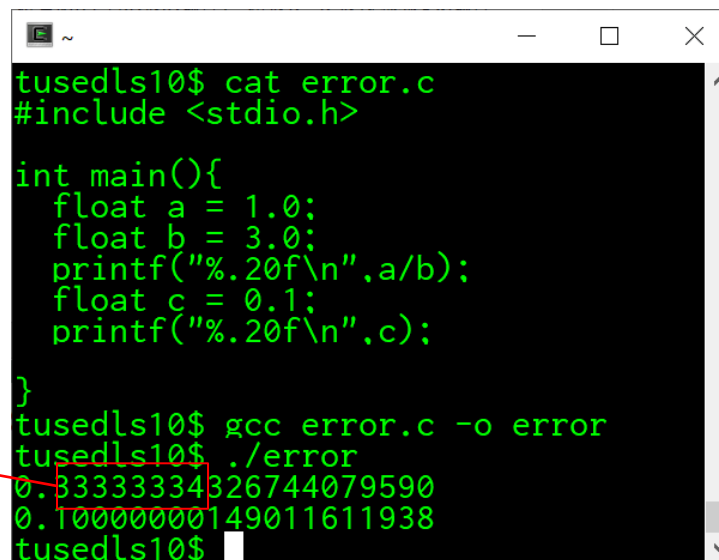
- $1.5(10\text{進数}) = 1.1(2\text{進数}) : 2^0 + 2^{-1}$
- $1.25(10\text{進数}) = 1.01(2\text{進数}) : 2^0 + 2^{-2}$

上記の数は表せる

- $0.1(10\text{進数}) = 0.0001100110011001100110011\dots$

無限に続く値は  
近似値でしか表せない

有効桁は6~7桁ぐらい



```
tusedls10$ cat error.c
#include <stdio.h>

int main(){
    float a = 1.0;
    float b = 3.0;
    printf("%.20f\n",a/b);
    float c = 0.1;
    printf("%.20f\n",c);
}
tusedls10$ gcc error.c -o error
tusedls10$ ./error
0.33333334326744079590
0.1000000149011611938
tusedls10$
```

# 具体例(10進数0.1)

```
tusedls10$ cat float2bit.c
#include <stdio.h>

union float_int {
    int i;
    float f;
};

int main(){
    int i=0;
    union float_int u;
    u.f=0.1;
    for(i=0;i<32;i++){
        if((u.i & 0x80000000) == 0x80000000){
            printf("1");
        }else{
            printf("0");
        }
        u.i = u.i<<1;
    }
    printf("\n");
}
tusedls10$ gcc float2bit.c -o float2bit
tusedls10$ ./float2bit
00111101110011001100110011001101
tusedls10$
```

符号部 0, 指数部01111011 (123),

仮数部 10011001100110011001101

→  $2^{(123-127)} * 1.10011001100110011001101$

→  $0.000110011001100110011001101$

→  $1/16 + 1/32 + 1/256 + 1/512 + 1/4096 + 1/8192...$

→  $0.0625 + 0.03125 + 0.00390625 + 0.001953125 + 0.000244140625 + 0.0001220703125...$

$1/16 + 1/32 + 1/64 = 0.109375 > 0.1$  になるので1/64のビットは使わない

# 浮動小数点の丸め

- 10進数であれば四捨五入で丸め処理
- 2進数であれば $2^n$ で丸める
  - 偶数丸め
  - 例:  $\frac{1}{2}$ で丸める場合
    - $10.010(2+1/4) = 10.0(2)$
    - $10.011(2+3/8) = 10.1(2+1/2)$
    - $10.110(2+3/4) = 11.0(3)$
    - $11.001(3+1/8) = 11.0(3)$
  - $10.010(2+1/4)$ は $10.0$  と $10.1$ の中間値  $\rightarrow 10.0$ にする(最下位ビットを0)
  - $10.110(2+3/4)$ は $11.0$  と $10.1$ の中間値  $\rightarrow 11.0$ にする(最下位ビットを0)

# 浮動小数点と整数の変換

int, float, double 間の変換(キャスト)は  
値を可能な限り保持する形で変換される  
丸める必要がある場合は偶数丸めが使用される

- int → float: 丸められるかもしれない
- int → double: 正確な値が保持できる (doubleの仮数部>32bit)
- float → double: 正確な値が保持できる
- double → float: 丸められるかもしれない
- float → int: 値は0の方へ丸められる(1.999→1, -1.999→-1)
- double → int: 同上

# まとめ

- コンピュータは情報をビットにエンコードし、ビットはバイト列として構成される
- 異なるエンコーディング(解釈の方法)が、数値、文字列、画像・・等に用いられる
- ビット列(ビットパターン)で情報が保持されており、ビット列を操作する方法
- エンコーディング(解釈の方法)を変えた際のビット列の変化



質問あればどうぞ