#03
基本型

2022 年度 / プログラミング及び実習 III

角川裕次

龍谷大学　先端理工学部

もくじ

今回 (#03) の内容：シラバスでの該当部分

小テーマ: **基本型**

第 5 回：**データ型と** sizeof **演算子**

重要概念リスト

- short, int, long
- signed と unsigned
- sizeof 演算子
- size_t 型
- typedef 宣言
- <limits.h> ヘッダ
- float, double, long double
- <math.h> ヘッダ
- 2 の補数表現
- IEEE 754 浮動小数点
- 演算子の優先度と結合性
- 型変換

今回の実習・課題 (manaba へ提出)

実習内容と課題内容は講義途中に提示します

(作成したファイル類は manaba **に提出)**

第 7-1 節　基本型と数

□ 算術型と基本型 p.182

算術型 (arithmetic type)：算術演算が定義されたデータ型

- 加算や乗算など
- `int` 型 や `double` 型などの変数や定数

基本型 (basic type)：型名キーワードだけで表せる型

- 文字型 (`char`)
- 整数型 (`int`)
- 浮動小数点型 (`double`)

算術型：多くの型の総称

Fig. 7-1 **算術型**

汎整数型

- **列挙型**
  enum～型
- **文字型**
  char 型
  signed char 型
  unsigned char 型
- **整数型**
  signed short int 型
  unsigned short int 型
  signed int 型
  unsigned int 型
  signed long int 型
  unsigned long int 型

浮動小数点型

- float 型
  double 型
  long double 型

□ 基数 p.183

数値を表す際の各桁の重み付けの基本となる数のこと

10 進数：わたしたちが日常使っている数の表記法
例：“西暦 2021 年”

- 「数」2021 を「文字 (数字)」を使って書き表している
- 使用する文字は 10 種類：0, 1, 2, ..., 9

コンピュータシステムを対象とする場合は 2 進数, 16 進数が便利

2 進数：数字 0 と 1 を使用 (2 種類)

- 表記の際の桁数は多い (1 桁が表現するのは 1 ビット)
- ビット単位で数値を表現するためハードウエア制御で便利

16 進数：数字 0, 1, ..., 9, A, ..., F を使用 (16 種類)

- 表記の際の桁数が少なくて済む (1 桁が表現するのは 4 ビット)
- メモリアドレスの表記など便利

数の表記の対応表

2 進数	10 進数	16 進数
0	0	0
1	1	1
10	2	2
11	3	3
100	4	4
101	5	5
110	6	6
111	7	7
1000	8	8
1001	9	9
1010	10	A
1011	11	B
1100	12	C
1101	13	D
1110	14	E
1111	15	F

2 進数	10 進数	16 進数
10000	16	10
10001	17	11
10010	18	12
10011	19	13
10100	20	14
10001	21	15
10110	22	16
10111	23	17
11000	24	18
11001	25	19
11010	26	1A
11011	27	1B
11100	28	1C
11101	29	1D
11110	30	1E
11111	31	1F

□ 基数変換 p.184

2 つの概念が微妙に混在：「文字 (数字) の列」と「数」

数：抽象概念

- 数の集合：無限集合 (数は無限通り存在)
- 数は任意に大きな値をとりうる

文字 (数字) の列：記号の列

- 使用する記号の集合：有限集合 (有限個数の記号を使用)
- 列は有限長

以下のように 2 つを区別 (カギ括弧を使用)

- 数：1234
- 文字の列：「1234」

(教科書では区別ができてないので注意)

文字の列から数への変換 (1)

例：10 **進数で書かれた文字の列「**1998**」が表す数を求める**

$$
\begin{aligned}
& 10\ \text{進数で「}1998\text{」が表す数} \\
&= \text{「}1\text{」} \times 10^3 + \text{「}9\text{」} \times 10^2 + \text{「}9\text{」} \times 10^1 + \text{「}8\text{」} \times 10^0 \\
&= 1 \times 10^3 + 9 \times 10^2 + 9 \times 10^1 + 8 \times 10^0 \\
&= 1998
\end{aligned}
$$

表す数は各桁の重み付きの和

- **第** i **桁には重み** 10^i (10 **進数**)
- **第** 0 **桁は右端の桁**

文字の列から数への変換 (2)

例：16 **進数で書かれた文字の列「**1FD**」が表す数を求める**

$$
\begin{aligned}
& 16\ \textbf{進数で「}1\mathrm{FD}\textbf{」が表す数} \\
= \quad & \textbf{「}1\textbf{」} \times 16^2 + \textbf{「}\mathrm{F}\textbf{」} \times 16^1 + \textbf{「}\mathrm{D}\textbf{」} \times 16^0 \\
= \quad & 1 \times 16^2 + 15 \times 16^1 + 13 \times 16^0 \\
= \quad & 509
\end{aligned}
$$

表す数は各桁の重み付きの和

- **第** i **桁には重み** 16^i (16 **進数**)

文字の列から数への変換 (3)

例：2 **進数で書かれた文字の列「**101**」が表す数を求める**

$$\begin{aligned}
&\text{2 進数で「101」が表す数} \\
&= \text{「1」} \times 2^2 + \text{「0」} \times 2^1 + \text{「1」} \times 2^0 \\
&= 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 \\
&= 5
\end{aligned}$$

表す数は各桁の重み付きの和

- **第** i **桁には重み** 2^i (16 **進数**)

数から文字の列への変換：考え方

例：509 **を** 16 **進数で書くと「**1FD**」**

$$
\begin{aligned}
509 &= (32 \times 16) + {\color{red}13} \\
&= 1 \times 16^2 + 15 \times 16^1 + 13 \times 16^0 \\
&= \text{「1」が表す数} \times 16^2 + \text{「F」が表す数} \times 16^1 + \text{「D」が表す数} \times 16^0 \\
&= \text{「1F」が表す数} \times 16^1 + \text{「D」が表す数} \\
&= \text{「1F}{\color{red}\text{D}}\text{」が表す数}
\end{aligned}
$$

観察ポイント:

数を 16 **で割った余りで**
16 **進数で書いたときの右端の桁の数字が分かる**

- k **進数の場合**：k **で割った余りで右端の数字が分かる**

数から文字の列への変換方法

パラメータ

- n : **表記したい数**
- b : **基数** (2, 8, 10, 16 **など**)

変換の手順

1. n **を** 16 **で割った余りを計算** : **右端から** 0 **番目の桁の数字が決まる**
2. n = n / 16
3. n **を** 16 **で割った余りを計算** : **右端から** 1 **番目の桁の数字が決まる**
4. n = n / 16
5. n **を** 16 **で割った余りを計算** : **右端から** 2 **番目の桁の数字が決まる**
6. n = n / 16

   ...

   n = 0 **になれば終了**

数から文字の列への変換の例

$n = 509$, $b = 16$ (16 進数) の場合

変換の経過

1. $n = 509$
2. n を 16 で割った余りは 13 : 右端から 0 番目の桁の数字は「D」
3. $n = n \ / \ 16 = 31$
4. n を 16 で割った余りは 15 : 右端から 1 番目の桁の数字は「F」
5. $n = n \ / \ 16 = 1$
6. n を 16 で割った余りは 1 : 右端から 1 番目の桁の数字は「1」
7. $n = n \ / \ 16 = 0$
   $n = 0$ になったので終了

16 進数表記は「1FD」

第 7-2 節　整数型と文字型

□ 整数型 (integer type) と文字型 (character type)

★重要★ p.186

有限範囲の連続した整数を表現する型

- **符号の有無により 2 種類に分類**

符号無し整数型 (unsigned integer type)

非負の整数を表現

- **型指定子** `unsigned` **を付ける**
- **変数宣言の例**：`unsigned int z;`

符号付き整数型 (signed integer type)

非負と負の整数を表現

- **型指定子** `signed` **を付ける (省略可)**
- **変数宣言の例**：`signed int y;`
- **変数宣言の例**：`int x;`

表現できる値の範囲による 4 種類の型

`char` `short int` `int` `long int`

(それぞれに符号付き版と符号無し版あり)

Fig.7-6 : **整数を表す型の分類**

	符号付き整数型		符号無し整数型
文字型	**signed char**	**char**	**unsigned char**
整数型	**signed short int** **signed int** **signed long int**		**unsigned short int** **unsigned int** **unsigned long int**

型の短縮名

型名を厳密に書くと入力の文字数が増えて面倒くさい
→ 短縮名の導入

- 例：正式な型名 `signed int` を短縮して `int` と書ける

Table 7-1：文字型・整数型の名称と短縮名

文字型	char			
	signed char			
	unsigned char			
整数型	signed short int	signed short	short int	short
	unsigned short int	unsigned short		
	signed int	signed	int	
	unsigned int	unsigned		
	signed long int	signed long	long int	long
	unsigned long int	unsigned long		

□ 整数型の使い分け p.188

int 型：実行環境で一番扱いやすく高速に演算できるビット数を採用

short 型：メモリサイズを節約したい時
long 型：広い範囲の数値を扱い時

- 処理系毎に異なる
- sizeof(short) $\leq$ sizeof(int) $\leq$ sizeof(long) に注意
- sizeof(short) $=$ sizeof(int) $=$ sizeof(long) の場合もあり

□ <limits.h> ヘッダ p.189

文字型と整数型の数値の範囲をマクロ記号で定義するヘッダ

型	最小値	最大値
char	CHAR_MIN	CHAR_MAX
unsigned char		UCHAR_MAX
signed char	SCHAR_MIN	SCHAR_MAX
short	SHRT_MIN	SHRT_MAX
int	INT_MIN	INT_MAX
long	LONG_MIN	LONG_MAX
unsigned short		USHRT_MAX
unsigned int		UINT_MAX
unsigned long		ULONG_MAX

重要：具体的な数値の定義は言語処理系で異なる場合あり

- 自分のところで調べた値が他所で通用するとは限らない
- 可搬性を考えたソースコードを書くことを心がけよう

Q. いくつかの最小値のマクロ記号が未定義なのはなぜ?

数値は処理系でいろいろ

教科書での想定

記号	値	備考
`INT_MIN`	−32768	`int` の最小値
`INT_MAX`	32767	`int` の最大値
`LONG_MIN`	−2147483648	`long` の最小値
`LONG_MAX`	2147483647	`long` の最大値

Ubuntu x86_64 (64 ビット) の cc

記号	値	備考
`INT_MIN`	−2147483648	`int` の最小値
`INT_MAX`	2147483647	`int` の最大値
`LONG_MIN`	−9223372036854775808	`long` の最小値
`LONG_MAX`	9223372036854775807	`long` の最大値

□ 文字型 p.190

文字を格納するための型

3 **通りの型**

`char` (**符号付きか符号無しかは処理系で異なる**)

`unsigned char` (**符号無し文字型**)

`signed char` (**符号付き文字型**)

□ ビットと CHAR_BIT　p.192

コンピュータ内でのデータ：ビット (bit) **の組み合わせで表現**

マクロ `CHAR_BIT`

- **文字型** `char` **が記憶域上で専有するビット数**
- **定義の 1 例**: `#define CHAR_BIT 8`

具体的な数値は処理系によって異なる (**ただし少なくとも** 8)

0	0	1	0	1	1	0	0

□ sizeof 演算子　p.192

★重要★

データの記憶に用いるメモリサイズ (バイト数) を得る演算子

例：int **型のバイト数を得る**

```
isize = sizeof(int);
```

- **値は処理系によって異なる**
- **ただし** char **型のサイズは必ず** 1(**バイト**)

存在意義：**処理系での実際のデータサイズを得る**

- **その値に応じたプログラムコードを書ける**
- **ポータビリティ (可搬性) のあるコードが書ける**
  - **ポータビリティ**：**ソースコードそのままで様々な環境で動作可能なこと**

C **言語の規格により以下の関係が成立**

$$\texttt{sizeof(short)} \leq \texttt{sizeof(int)} \leq \texttt{sizeof(long)}$$

★重要★

□ size_t 型と typedef 宣言 p.194

typedef 宣言：既存の型の同義語を作る

例：新たに size_t 型を作る (unsigned 型と同義)

```
typedef unsigned size_t;
```

例：定義した size_t 型で変数を宣言

```
size_t isize = sizeof(int);
```

一般的な構文

```
typedef 型A 型B;
```

- 型 A：既存の型名
- 型 B：新たに定義する型 (型 A と同義)

□ 配列の要素数の求め方 p.196

List 7-5

```
int main(void) {
  int a[5];
  double x[7];
  printf("配列aの要素数=%zu\n", sizeof(a)/sizeof(a[0]));
  printf("配列xの要素数=%zu\n", sizeof(x)/sizeof(x[0]));
  return 0;
}
```

説明

- sizeof(a)
  配列 a 全体のバイト数
- sizeof(a[0])
  配列 a の 1 要素のバイト数
- sizeof(a)/sizeof(a[0])
  配列 a の要素数

★重要★

□ 符号無し整数の内部表現 p.198

ビットパターン $B_0, B_1, \ldots$ が表す数値 n

$$n = \sum_{i=0}^{n-1} B_i \times 2^i$$

- B_i : **第 i ビットの値** (0 **または** 1)
- **第 i ビットの重みは** 2^i

$B_{n\text{-}1}$	$B_{n\text{-}2}$	$B_{n\text{-}3}$	....	B_3	B_2	B_1	B_0

符号無し整数 $n = \sum_{i=0}^{n-1} B_i \times 2^i$

16 ビットの場合の例

ビット番号	15	14	13	…………	3	2	1	0
重み	2^{15}	2^{14}	2^{13}	…………	2^3	2^2	2^1	2^0
ビット	B_{15}	B_{14}	B_{13}	…………	B_3	B_2	B_1	B_0

上位ビット　　　下位ビット

例： `unsigned` **型の値** 25 **のビット表現**

ビット番号

15	14								6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	1

上位ビット　　　下位ビット

★重要★

□ 符号付き整数の内部表現 p.200

3 通りあり

- 符号と絶対値
- 1 の補数
- 2 の補数

2 の補数表現

- 多くの処理系で採用
- n ビットを使用するとき数値の範囲：-2^{n-1} から $2^{n-1}-1$

★重要★

2 の補数表現

ビットパターン $B_0, B_1, \ldots$ が表す数値 n

$$n = -B_{n-1} \times 2^{n-1} + \sum_{i=0}^{n-2} B_i \times 2^i$$

- B_i : **第 i ビットの値** (0 **または** 1)

第 B_{n-1} ビット : **符号ビットと呼ばれる**

- **符号ビットが** 1 **のとき** : **負の値**
- **符号ビットの重み** : -2^{n-1}

$B_{n\text{-}1}$	$B_{n\text{-}2}$	$B_{n\text{-}3}$	....	B_3	B_2	B_1	B_0

符号付き整数 $n = -B_{n-1} \times 2^{n-1} + \sum_{i=0}^{n-2} B_i \times 2^i$

16 ビットの場合の例

ビット番号	15	14	13	....	3	2	1	0
重み	-2^{15}	2^{14}	2^{13}	....	2^3	2^2	2^1	2^0
ビット	B_{15}	B_{14}	B_{13}	....	B_3	B_2	B_1	B_0

← 上位ビット　　　　下位ビット →

例：2 の補数表現による値 -25 のビット表現

ビット番号

15	14								6	5	4	3	2	1	0
1	1	1	1	1	1	1	1	1	1	1	0	0	1	1	1

← 上位ビット　　　　下位ビット →

□ ビット単位の論理演算 p.202

2 整数の各第 i ビット同士での論理演算

ビットごとの AND (論理積) 演算子 &

a =	0	0	0	0	0	1	1	1	1	0	1	0	1	0	1	1
b =	0	0	1	1	0	0	0	0	0	0	1	1	1	0	0	1
a&b =	0	0	0	0	0	0	0	0	0	0	1	0	1	0	0	1

ビットごとの OR (論理和) 演算子 |

a =	0	0	0	0	0	1	1	1	1	0	1	0	1	0	1	1
b =	0	0	1	1	0	0	0	0	0	0	1	1	1	0	0	1
a|b =	0	0	1	1	0	1	1	1	1	0	1	1	1	0	1	1

ビット単位の論理演算 (つづき)

ビットごとの XOR(**排他的論理和**) **演算子** ^

a =	0	0	0	0	0	1	1	1	1	0	1	0	1	0	1	1
b =	0	0	1	1	0	0	0	0	0	0	1	1	1	0	0	1
a^b =	0	0	1	1	0	1	1	1	1	0	0	1	0	0	1	0

ビットごとの NOT (**論理否定**) **演算子** ~

a =	0	0	0	0	0	1	1	1	1	0	1	0	1	0	1	1
~a =	1	1	1	1	1	0	0	0	0	1	0	1	0	1	0	0

□ シフト演算　p.204

左シフト

a << b : a を b ビット左へシフトする

例：4 ビット左へシフト

a =	0	0	0	0	0	1	1	1	1	0	1	0	1	0	1	1
a<<4 =	0	1	1	1	1	0	1	0	1	0	1	1	0	0	0	0

- はみ出したビットは捨てられる
- 右からは 0 が入る

性質：1 ビットシフトした値は元の値の 2 倍

- (オーバーフローしなければ)

シフト演算：右論理シフト

整数が符号無し/符号ありで論理シフト/算術シフトの区別あり

右論理シフト

a >> b：a を b ビット右へ (論理) シフトする

例：4 ビット右へシフト

```
unsigned int a = | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 1 |
          a>>4 = | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 0 | 1 | 0 |
```

- はみ出したビットは捨てられる
- 左からは 0 が入る

右論理シフトの性質：1 ビットシフトした値は元の値の 1/2 倍

- (小数は切り捨て)

シフト演算：右算術シフト

例：4 ビット右へ (算術) シフト (正の数の場合)

`signed int a =`	0	1	0	0	0	1	1	1	1	0	1	0	1	0	1	1
`a>>4 =`	0	0	0	0	0	0	1	0	0	1	1	1	1	0	1	0

例：4 ビット右へ (算術) シフト (負の数の場合)

`signed int a =`	1	0	0	0	0	1	1	1	1	0	1	0	1	0	1	1
`a>>4 =`	1	1	1	1	1	0	0	0	0	1	1	1	1	0	1	0

右算術シフトの性質：1 ビットシフトした値は元の値の 1/2 倍

- 重要点：正負の符号も保存
- (小数は切り捨て)

シフト演算：右算術シフト (つづき)

右算術シフト

`a >> b`：`a` を `b` ビット右へ (算術) シフトする

- はみ出したビットは捨てられる
- 符号ビットが 0 のとき：左からは 0 が入る
- 符号ビットが 1 のとき：左からは 1 が入る

例：整数中の 1 のビット数を勘定

List 7-6 (部分)

```
int count_bits(unsigned x)
{
  int bits = 0;
  while (x) {
    if (x & 1U) bits++;
    x = x >> 1;
  }
  return bits;
}
```

- while 文：`x` が 0 ならもう `x` には値が 1 のビットはないので終了
- `if (x & 1U)`：`x` の第 0 ビットが 1 ならカウント (`bits++`)
- `1U`：符号無し整数 1 を表す
- `x = x >> 1`：`x` を 1 ビット右へシフト

教科書の内容 `x >>= 1;` は `x = x >> 1;` と同等

bit_count の動作の図解

例：整数 (int) 型の表現で使用するビット数を得る

List 7-6 (**部分**)

```
int int_bits(void)
{
  return count_bits(~0U);
}
```

- ~0U：0 **の全ビットの反転**... int **型のビット数だけ** 1 **のビットあり**
- count_bit：1 **のビット数を勘定**

16 **ビットでの例**

ビット番号

	15	14									6	5	4	3	2	1	0
0U	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
~0U	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

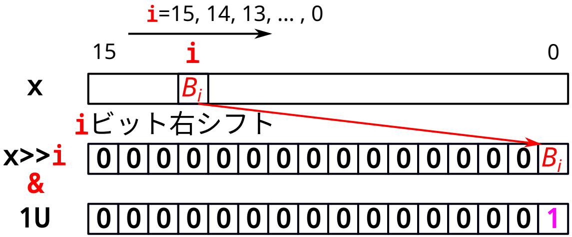
count_bits(~0U) ⟶ 16

例：符号無し整数 x のビットパターンを表示

List 7-6 (**部分**)

```
void print_bits(unsigned x)
{
  for (int i = int_bits() - 1; i >= 0; i--)
    putchar(((x >> i) & 1U) ? '1' : '0');
}
```

最上位ビットから順次表示

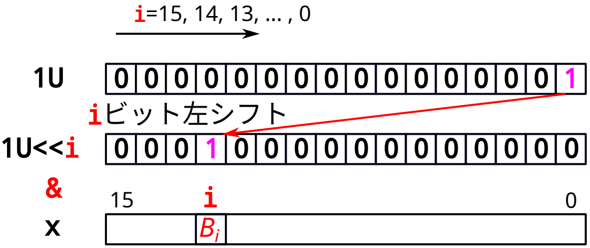


`((x >> i) & 1U)` **は** `((x >> i) & 1U) != 0` **と同等**

別実装：符号無し整数 x のビットパターンを表示

```
void print_bits2(unsigned x)
{
  for (int i = int_bits() - 1; i >= 0; i--)
    putchar((x & (1U << i)) ? '1' : '0');
}
```

最上位ビットから順次表示



`(x & (1U << i))` **は** `(x & (1U << i)) != 0` **と同等**

□ ビット単位の論理演算の応用 p.210

論理和: **任意に指定されるビットをセット** (1 **にする**)

例: `n|1U`

- `1111111111100000` : n **の二進表現** (**例**)
- `0000000000000001` : `1U` **の二進表現** (**指定ビット**)
- `1111111111100001` : `n|~1U` **の二進表現**

論理積: **任意に指定されるビットをリセット** (0 **にする**)

例: `n&~1U`

- `1111111111100011` : n **の二進表現** (**例**)
- `1111111111111110` : `~1U` **の二進表現** (**指定ビット**)
- `1111111111100010` : `n&~1U` **の二進表現**

排他的論理和: **任意に指定されるビットを反転** (0 **を** 1 **に**, 1 **を** 0 **に**)

例: `n^1U`

- `1111111111110101` : n **の二進表現** (**例**)
- `0000000000011111` : `1U` **の二進表現** (**指定ビット**)
- `1111111111101010` : `n|~1U` **の二進表現**

□ 整数定数 p.210

ソースコード中での書き方いろいろ

8 **進定数**

- **先頭に** 0 **を付ける**
- **例**： 013 (10 **進数では** 11)

10 **進定数**

- **先頭に** 0 **を付けない**
- **ただし値** 0 **の場合は例外**

16 **進定数**

- **先頭に** 0x **を付ける**
- **例**： 0x12 (10 **進数では** 18)

2 **進定数** (**非標準**)

- **先頭に** 0b **を付ける**
- **例**： 0b01011 (10 **進数では** 11)
- **組込みシステム系でよく使用**

□ 整数定数の型 p.211

整数接尾辞：整数定数の型の明示する方法

`U` **または** `u`

- **符号無し** (unsigned) **であることを明示**
- **例**：`3517U`

`L` **または** `l`

- long **型であることを明示**
- **例**：`127569L`

□ 整数の表示 p.212

printf は書式指定により 8 進数, 10 進数, 16 進数の表示が可能

8 進数 (octal)

- `%o`
- 例：`printf("%06o\n", n);`

10 進数 (decimal)

- `%d`：int 型
- `%ld`：long 型
- 例：`printf("%04d\n", n);`

16 進数 (hexadecimal)

- `%x`：int 型
- `%lx`：long 型
- 例：`printf("%08x\n", n);`

★重要★

□ オーバーフローと例外 p.213

オーバーフォロー (overflow)：

演算結果が表現可能な範囲を超えること (**桁あふれ**)

- **決まったビット数で表現しているため**

期待していたものと違う結果を得るので要注意

例：32 **ビット符号付き整数の減算の場合**

	ビット番号	31	30	29	28		……………………		3	2	1	0
x =	1	0	0	0	0	……………………	0	0	0	0	0
	-2147483648										
x-1 =	0	1	1	1	1	……………………	1	1	1	1	1
	2147483647										

結果は負になるはずが正の値に

オーバーフロー (つづき)

★重要★

例：32 **ビット符号付き整数の加算の場合**

ビット番号	31	30	29	28	…………………	3	2	1	0
x =	0	1	1	1	1 ………………… 1	1	1	1	1

2147483647

x+1 =	1	0	0	0	0	…………………	0	0	0	0	0

-2147483648

結果は正になるはずが負の値に

第 7-3 節　浮動小数点型

□ 浮動小数点型 p.214

小数部を持つ実数を表現

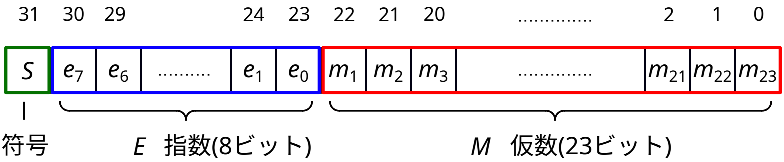
3 種のデータ型

- `float`
- `double`
- `long double`
- **それぞれに何ビットを使って表現するかは処理系によって異なる**

宣言, 定数, 表示

```
float p        = 3.1415F;
double g       = 6.67430E-11;
long double e = 1.6021766E-19L;
printf("%f\n",  p);
printf("%f\n",  g);
printf("%Lf\n", g);
```

IEEE 754 浮動小数点数 (単精度) 規格



ビットパターンが表現する値 r

$$
\begin{aligned}
r &= (-1)^S \times M \times 2^E \\
S &= B_{31} \\
M &= 1 + \sum_{i=1}^{23} 2^{-i} \times m_i \quad = \quad 1 + \sum_{i=0}^{22} 2^{-i} \times B_{22-i} \\
E &= \left(\sum_{i=0}^{7} 2^i \times e_i \right) - 127 \quad = \quad \left(\sum_{i=0}^{7} 2^i \times B_{23+i} \right) - 127
\end{aligned}
$$

□ 浮動小数点定数 p.216

定数として数値そのものをプログラムコード中に書く方法

型を明示する方法 (無指定 / f, F / l, L)

- `57.3` — double 型
- `57.3F` — float 型
- `57.3L` — long double 型

指数表記

- `1.23E4` — 1.23×10^4 を表す
- `89.3E-5` — $89.3 \times 10^{-5} = 8.93 \times 10^{-4}$ を表す

その他の例

- `.5` — double 型 0.5 を表す
- `12.` — double 型 12.0 を表す

★重要★

□ <math.h> ヘッダ：各種の数学関数を宣言 p.217

みなさん頻繁に使うはず

- sqrt (**平方根**)
- fabs (**絶対値**)
- cos, sin, tan (**三角関数：余弦, 正弦, 正接**)
- log, log10 (**対数**)
- exp, pow (**指数**)
- ceil, floor (**天井, 床**)
- acos, asin, atan, atan2 (**逆三角関数**)
- cosh, sinh, tanh
- frexp, modf
- ldexp, fmod

List 7-11 : **平面上の 2 点間の距離を計算** p.217

```
#include <math.h>
#include <stdio.h>

/*--- 点(x1,y1)と点(x2,y2)の距離を求める ---*/
double dist(double x1, double y1, double x2, double y2)
{
  return sqrt((x2 - x1) * (x2 - x1) +
              (y2 - y1) * (y2 - y1));
}

int main(void)
{
  double x1, y1;  /* 点1 */
  double x2, y2;  /* 点2 */
  printf("2点間の距離を求めます。\n");
  printf("点1…X座標："); scanf("%lf", &x1);
  printf("     Y座標："); scanf("%lf", &y1);
  printf("点2…X座標："); scanf("%lf", &x2);
  printf("     Y座標："); scanf("%lf", &y2);
  printf("距離は%fです。\n", dist(x1, y1, x2, y2));
  return 0;
}
```

解説

ヘッダのインクルード (数学関数用)

```
#include <math.h>
```

sqrt : **平方根** (square root) **関数**

```
return sqrt((x2 - x1) * (x2 - x1) +
            (y2 - y1) * (y2 - y1));
```

scanf **での** double **型データの読み込み**

- **注意点** : `%f` **ではなく** `%lf`

```
scanf("%lf", &x1);
```

□ 繰り返しの制御 p.218

List 7-12 (**部分**) **コード悪例** : 0.0 **から** 1.0 **まで** 0.01 **単位で繰り返す**

```
float x;
for (x = 0.0; x  <= 1.0; x += 0.01) {
  printf("x=%f\n", x);
}
```

出力 (Ubuntu 16.04, x86_64, cc)

```
x=0.000000
x=0.010000
    ...  略  ...
x=0.989999
x=0.999999
```

困った現象 : 100 **回繰り返したけど** 1.00 **になっていない**

理由 : **誤差の蓄積**

- 10 **進数では** 0.01 **は切りのいい数値**
- **でもコンピュータ内部は** 2 **進数で数値を表現**
- 2 **進数では** 0.01 **は切りが良くない数値** (**誤差あり**)
  → **僅かな誤差が蓄積**

誤差を少なくするには

解決法：実数値をループの制御には使わない; 整数値を使う

List 7-13：**繰り返し制御を整数で行う**

```
#include <stdio.h>

int main(void)
{
  float x;
  for (int i = 0; i <= 100; i++) {
    x = i / 100.0;
    printf("x = %f\n", x);
  }
  return 0;
}
```

出力：誤差の蓄積が生じない

第 7-4 節　演算と演算子

★重要★

□ 演算子の優先順位と結合性 p.220

優先順位

ひとつの式の中でどの演算子を先に演算するかを規定

- **例**：a + b * c **は乗算を先に計算**

結合性

同じ優先度の演算子を続けて書いた場合の計算順序を規定

- **左結合**：**式** a ○ b ○ c **を** (a ○ b) ○ c **と計算**
- **右結合**：**式** a ○ b ○ c **を** a ○ (b ○ c) **と計算**

- **例 (減算)**：a - b - c **は** (a - b) - c **の順で計算**
- **例 (代入)**：a = b = 1 **は** a = (b = 1) **の順で計算**

演算子の一覧 (優先順位と結合性).... Table 7-11 **参照**

□ 型変換の規則 p.222

背景：型が違うと表現できる値の範囲が異なる場合がある

- 型が違うとデータ表現に用いるビット数が異なる場合があるため

型変換を気にする理由：変換が正しく行われるとは限らない

- 変換後のデータ型の値の範囲が狭いと変換結果が正しくない場合あり
- 例 1: int 型の値 4096 を char 型へは正確には変換できない (たぶん)
- 例 2: double 型の値 3.1415 を int 型へは正確には変換できない

C 言語では型変換の規則を細かく規定

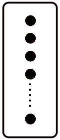
- 変換後の結果を正確に規定するため

型変換を「写像」と考えるとわかりやすい

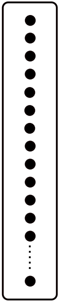
- 以降の説明：教科書に書いてあることの背景

ビット数と表現可能な数の個数の関係

8 **ビット**：2^8 **通り**



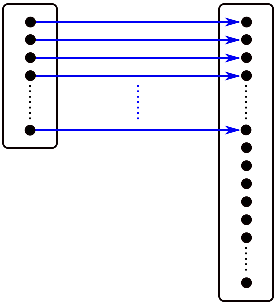
32 **ビット**：2^{32} **通り**



- **各●は値**
- **枠線は集合を表す**
- **左図**：**要素数** $2^8 = 256$ **の集合**
- **右図**：**要素数** 2^{32} **の集合**

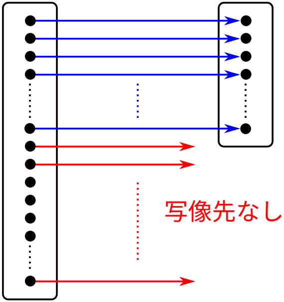
8 ビットのデータ型の数値 → 32 ビットのデータ型へ

正しい変換が可能
理由：変換前のどの値にも写像先が存在するため



32 ビットのデータ型の数値 → 8 ビットのデータ型へ

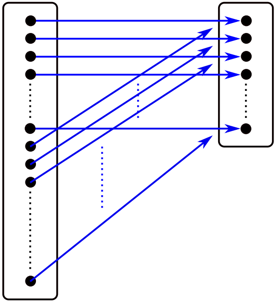
正しい変換は不可能
理由：変換前の一部の値にしか写像先が存在しないため



変換後で用いるビット数が不足

8 ビットでは表現できない値が存在

変換後のビット数が不足の時には?
— 多対 1 (many-to-one) 写像



例 1： 32 **ビット整数を** 8 **ビット整数へ変換**

- 256 **を変換すると** 0 **へ**
- 257 **を変換すると** 1 **へ**

例 2： 64 **ビット浮動小数点数を** 8 **ビット整数へ変換**

- 3.14 **を変換すると** 3 **へ**
- 3.2 **を変換すると** 3 **へ**

正しくない値に変換される結果となる場合あり

おわり