コンピュータとプログラミング C言語編

2. データ表現

阿部洋丈 / ABE Hirotake
habe@cs.tsukuba.ac.jp

本題の前に： C言語の基本（Pythonとの対比を通じて）

- Python との主な相違点：
  - プログラムは必ず関数として書く（mainから実行される）
  - 変数を使う場合は宣言が必要
  - ブロック（文のかたまり）を表現するためには {...} を使う。インデントは無視される
  - 文の末尾にはセミコロン（；）が必要
  - リスト、タプル、辞書は言語レベルでは提供されていない。for ループは変数を使って回すことが一般的。

簡単なプログラムの比較

```
sum = 0
for i in range(11):
    sum += i
print(sum)
```

```
#include <stdio.h>
int main(void) {
    int i;
    int sum = 0;
    for (i = 0; i <= 10; i++) {
        sum += i;
    }
    printf("%d\n", sum);
    return 0;
}
```

関数として書く

変数宣言が必要

{…} で囲む

リストを使って
ループを回す
ことはできない

インデントは意味を持たない
（読みやすさのために入れる）

セミコロン

C言語で使える制御構文

- if (条件式) {文;...} else {文;...}
  - if (条件式) {文;...} else if (条件式) {文;...} else {文;...} とも書ける
- for (初期化; 条件式; 更新) {文;...}
- while (条件式) {文;...}
- do {文;...} while (条件式); ←セミコロンを忘れずに
- break や continue も使える
- goto（ほぼ jmp そのもの）もあるが、使用は推奨されない

for の動作についてより詳しく

- for は Python のものと結構違うので少し注意が必要

```
for (i = 0; i <= 10; i++) { printf("hello"); }
```

gcc -S

```
main:
.LFB0:
        （中略）
        movl    $0, -4(%rbp)
        jmp     .L2
.L3:
        movl    $.LC0, %edi
        movl    $0, %eax
        call    printf
        addl    $1, -4(%rbp)
.L2:
        cmpl    $10, -4(%rbp)
        jle     .L3
        （以下略）
```

i を初期化

.L2 へ飛ぶ

hello を表示

i を更新

継続条件を評価

switch --- Python には無い便利な構文

```
switch（式）{
    case 値1:
        文; ...
        break;
    case 値2:
        文; ...
        break;
    default:
        文; ...
}
```

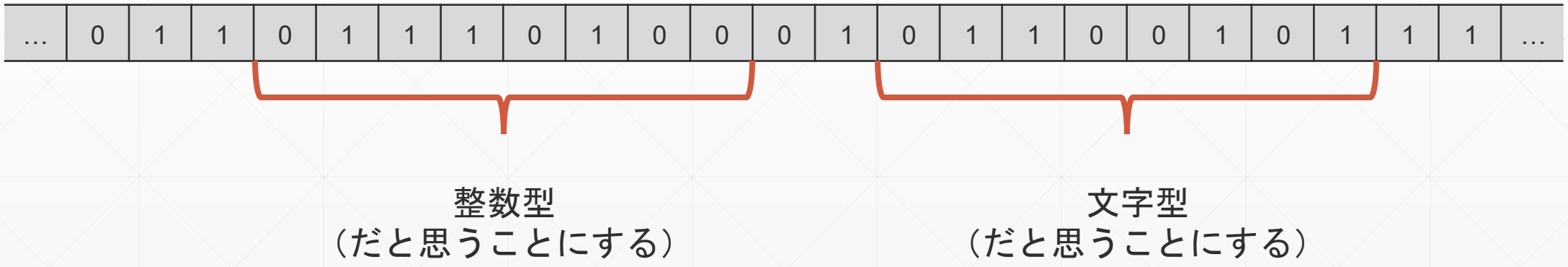
- 「式」の値を計算し、それと一致する case ラベルにジャンプする
  - どれとも一致しなければ default へジャンプ
  - break は入れなくてもOK。その場合はそのまま次の case へ進む
- if 文を一般化したような構文
  - 同じことを Python でやろうとすると面倒（break が必ず毎回入るなら簡単だが…）
- アセンブリ言語的なコードが簡単に書ける

演算子についての補足 (Python との比較)

- 算術演算子や代入演算子： 下記以外は大体同じ
  - 整数同士の "/" は、Python の "//" として働く（商のみが返される）
  - "**"は使えない （別の意味になるので注意）
- 比較演算子： in が使えない以外は大体同じ
- ビット演算子： ほぼ同じ
- 論理演算子： and は "&&" 、or は "||" と書く
  - true や false は無い。0 を false、それ以外の整数を true として扱う
- 3項演算子： "*a* if *c* else *b*" は "*c* ? *a* : *b*" と書く

型(type)とは

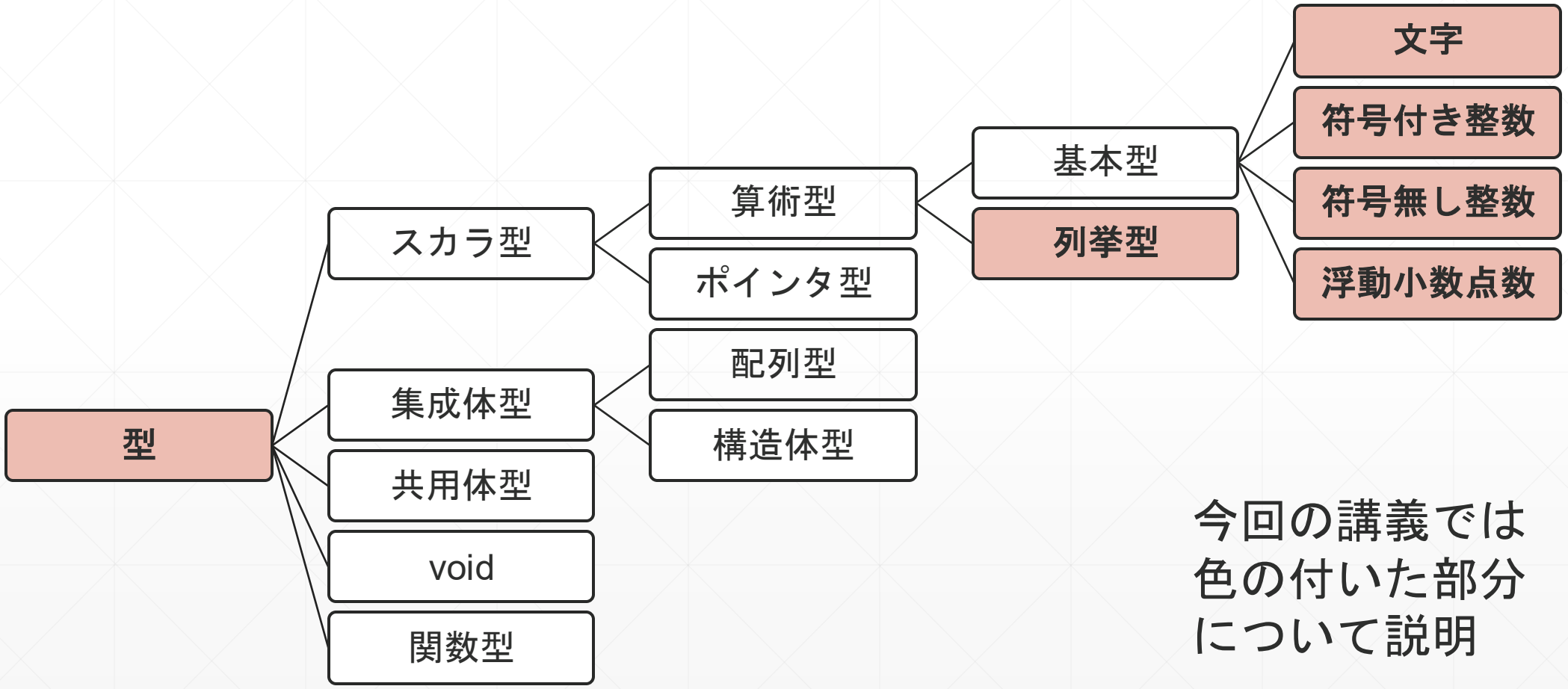
- あるビット列がどのようなデータを表現しているかを示すラベルのようなもの。
  - 必ずしも明示的にラベルがついているわけではない。コンパイル後は不要になることが多い。



なぜ型が必要か

- ビット列を見てもそれが何のデータかは分からない
  - ある型のデータを無理矢理別のデータとして解釈することも出来てしまう。（例： 整数と実数（浮動小数点数））
- 異なる型でビット列を解釈するとプログラムが誤動作する恐れがある（例： 符号付きと符号なし）
  - 誤った型でアクセスしていることをヒントにプログラムのバグを発見出来ることもある

C言語における型の一覧



今回の講義では
色の付いた部分
について説明

整数型

- アセンブリ言語で扱う整数そのもの（なので詳細は省略）
- {signed, unsigned} x {short int, int, long int} の6種類
  - 型名にはいくつかの省略が許されている
    - signed とも unsigned とも書いてない場合は signed と見なされる
    - short int は **short**、long int は **long**、signed int は **signed**、unsigned int は **unsigned** だけでも良い。
  - coins 環境では、short, int, long はそれぞれ 16bit, 32bit, 64bit
- とりあえず long を使っておけば桁あふれの恐れは減る
  - 適切に型を選ばないとメモリが溢れる場合もあるので注意

整数についてさらに詳しく

- 厳密に言えば、6種類の整数はそれぞれ異なる型
  - 原理主義的には、これらは混在させられない
  - 数字の列の末尾に目印をつけることで、その整数の型を指定することができる。
  - Uを付けると unsigned int。Lを付けると signed long。等々。
- ただ、厳密にこれを守ろうとすると面倒くさい
  - int 型の変数に 10L や 10U は代入できないの…？
  - 10 と 10L は足し算できないの…？

暗黙の型変換

ここで言う「大きい」「小さい」は、表現するのに必要なビット数のこと。それで表現されている値自体の話ではないことに注意

- いくつかのケースでは自動的に型を変換した上で処理が進められる
  - **整数拡張**： int より小さい整数型は、 int もしくは unsigned に変換されて処理される
  - **通常の算術型変換**： 異なる型同士の算術演算は、大きい方の型に揃えられる
  - **代入および関数呼び出しによる変換**： 代入先の変数の型に合わせて変換される。より小さい型への変換も起きるので注意が必要！

注意が必要なケースの例

- 異なる型同士の演算結果を小さい方の型に格納

```
int i = 1;
long l = 2147483647;  // int で表現できる最大の値
i = i + l;    // i には 2147483648 ではない値が入る
```

1. 2行目の代入で、2147483647 が 2147483647L に変換される。
2. 3行目の加算を実行する際、i は long に変換され、加算の結果も long になる。
3. 加算の結果である long の値を int である i に格納する際にビット数が足りなくなり、期待しない値が入ってしまう。

明示的な型変換（型キャスト）

- 副作用を十分理解した上で、明示的に型変換を起こさせることも可能。
- 変数名や定数名の前に、カッコで囲んだ型名を書いておくとその型に変換される。

```
int i = 2147483647;
int j = (signed short) 2147483647;
int k = (unsigned short) 2147483647;
```

i, j, k にはそれぞれ異なる値が入る

- 整数と実数が混在する場合は必要になる場合がある。（くわしくは後述）

文字型

- 文字を整数に対応させて扱う
  - 一般に、機械語には文字を直接扱う命令は無い
  - 整数<->文字のためのコード体系が ASCII や Unicode
- C言語では、英字（数字や記号も含む）を表現するために char 型を用いる。
  - 実装的には short よりもさらに小さい整数型。一般的には 8 bit で表現されている。unsigned char もある。

ASCII
(American Standard Code for Information Interchange)

- 計算機や通信でよく使われる文字コード体系の一つ。
  7bit で 1 つの文字を表現。
  - 0〜31、127： 制御文字（改行など）
  - 32： 空白文字（スペース）
  - 48〜57： 数字 (0123456789)
  - 65〜90： 英大文字 (ABC...XYZ)
  - 97〜122： 英小文字 (abc...xyz)
  - 上記以外の部分： 各種記号（!, ", #, $, ...）

本授業では Unicode 等のmultibyte文字は扱わない。興味のある人は各自で調べること。

文字を扱うプログラムの例

- 大文字と小文字の変換（敢えて冗長に書いてある）

```
char c = 'A';    // 文字を表現したい場合はシングルクウォートで囲む
putchar(c);      // 画面に A が表示される
int i = c;       // 暗黙の型変換により int 型に変換される
i += 32;         // i に 32 が加算される
c = (char) i;    // int 型の値を char 型にキャスト
putchar(c);      // 画面に a が表示される
```

- これの逆をやれば小文字から大文字への変換が出来る
- 実際にこのようなコードを書く場合は、入力の値が正しい範囲内かを検査することが必要。

浮動小数点数型

- 実数全体を表現するためには無限のビット長が必要
  - 整数を基本とした計算機で扱うことは事実上不可能
- そのため、本物の実数の代わりに浮動小数点数（floating point number）が用いられている

$$(-1)^S \times B^E \times F$$

Sは1ビット。
Bの値、および、E、F のビット長は規格毎に異なる

- IEEE754 規格
  - よく使われている浮動小数点数規格。基数 B は 2。
  - 単精度（E:8bit, F:23bit）や倍精度（E:11bit, F:52bit）などが定義されている。

浮動小数点数の型

- float 型（単精度）、double 型（倍精度）、long double 型（拡張倍精度）がある。
  - IEEE754 が使われることが一般的だが、アーキテクチャによっては異なる可能性がある。
  - signed, unsigned の区別はしない。（少なくとも言語規格上は）
- 表現可能な値の範囲（正規化数の絶対値）
  - float: 1.175×10^{-38} ～ 3.402×10^{38}
  - double: 2.225×10^{-308} ～ 1.798×10^{308}

正規化数(normalized number)：
F(仮数部)が $1.F_0F_1F_2...$ で表現されている数。
こうした方が誤差が少なくなる(例：$2^1 \times 1.0101$ v.s. $2^2 \times 0.1010$)
ただし、値が小さすぎるとこの方法では表現できなくなる（非正規化数; subnormalized number)

浮動小数点数を使う場合の注意(1)

- 誤差が避けられない
  - 丸め誤差のために、0.1 ですら誤差なしには表現できない
    （0.1 は 0.1000000149011612）
  - 計算方法次第で更に誤差が拡大することがある
    - 桁落ち： 1.23456 – 1.23444 の結果が 0.00012333333 に
    - 情報欠落： 77777.7 + 1.23456 の結果が 77778.93769 に
    - 無限級数などを扱う場合は打切り誤差も問題になる
  - 計算を繰り返すことで誤差がさらに拡大することもある

浮動小数点数を使う場合の注意(2)

- 整数と浮動小数点数の混在に注意
  - 整数と浮動小数点数の演算は、整数を浮動小数点数に変換した上で実行される (暗黙の算術型変換)
  - 整数の除算で浮動小数点数を得たい場合には暗黙の型変換や型キャストを利用する必要がある

```
float f1 = 1 / 2;                    // 0.000000...
float f2 = (float) 1 / (float) 2;    // 0.500000...
float f3 = (float) (1 / 2);          // 0.000000...
float f4 = (float) 1 / 2;            // 0.500000...
```

列挙型

- 予め決められた値のみをとる型をプログラマ自身が定義することができる。その実体は int 型の変数。

```
enum cardsuit { CLUBS, DIAMONDS, HEARTS, SPADES };

enum cardsuit suit;
suit = CLUBS;

if (suit == CLUBS) {
    ...
}
```

- プログラムを読みやすくするために利用可能
  - true や false が欲しい場合は： "enum boolean { FALSE, TRUE };"

const 修飾子

- 変数定義の先頭に const と付けておくと、その変数の変更ができなくなる（コンパイラが変更を防ぐ）
  - プログラムのバグを減らすことができる

```
double area_of_circle(double radius) {
  const double pi = 3.1415926535;

  pi = 3.0;   // コンパイルエラーが出る（かなり深刻な誤り）

  return pi * radius * radius;
}
```

今日の演習

ヒント： 指数部は "01111111" が 0 に相当する

- 0, 1, 2, ..., 7 の unsigned、および、0.0, 1.0, 2.0, ..., 7.0 の float の値について、それを表現するビット列を画面に表示するプログラムを作成せよ。
  - 変数 val に格納された値の、上位からn番目のビットを取り出すには以下の方法を利用すると良い。

```
*((unsigned *) &val) >> (sizeof(val) * 8 - n - 1) & 1
```

  - 上記の式を実行すると、0もしくは1の整数が得られる。それに応じてputchar等で画面に文字を表示すれば良い
  - 上記の式の意味は後日説明
- 1.0, ... 7.0 について、そのビット表現を解読し、確かにその値を表していることを計算によって確認せよ。