計算機言語 I 第 3 回
レポートに対するコメント, 教科書　2 章後半部分への補足

この資料: http://www.math.u-ryukyu.ac.jp/~suga/gengo/2021/03.pdf

1　2.6 章: 数学関数への補足

ここに書いてある内容は, Linux, macOS X などの Unix 由来の開発環境に対するものです. Windows の Visual Studio は, これらとは全く異なる開発環境なので, どうなっているかは調べないとわかりませんし, 私は調べてません.

sin, cos などの三角関数, 指数関数, 対数関数などのよく利用される初等関数は, 「数学関数ライブラリ」という形で, ライブラリにまとめられています.

これらを利用するには, `#include <math.h>` で，数学関数の定義を書いたヘッダファイルを取り込む必要があます. コンパイルの際には, 数学関するライブラリを利用することをコンパイラ (実際にはリンクエディタ) に知らせるためのオプション `-lm` が必要です（この部分は, Visual Stdio では異なる). これらは, 初回の講義 (教科書, 第 1 章) で述べた通りです.

Linux などの Unix 由来の開発環境で,「どのような数学関数がライブラリにあるか?」などを調べるには, `man` コマンドが利用できます.

```
bash-4.4$ man math.h
```

実行してみると, さまざまな数学定数と数学関数の定義が書かれています. 教科書 p. 28, 表 2.3 以外にも, いろいろな関数があります. 例えば, `j0` という関数が定義されていますが,

```
bash-4.4$ man j0
```

を実行すると, それが第 1 種 Bessel 関数を計算するものであることがわかります.

教科書 p. 27, プログラム 2.4 は, `M_PI` を利用してプログラムを書いて下さい.

2　変数の型: 教科書 2.8, 2.9 節の補足

この辺のことをきちんと理解しようとすると, C の歴史, 規格, 実装等を知るのが早いので, それを少し解説します.

2.1　CPU の歴史と　C

コンピュータで処理を実行しているのは, CPU(Central Processing Unit, 中央処理装置) です. CPU 内部には, レジスタ (register) と呼ばれる部分があり, そこで, 加法や乗法の計算を実行します. レジスタの考え方はそろばんと同じで, 有限桁しか扱えないところは全く一致しています. 現実のそろばんと違うのは, 2 進法の計算しかできないところです.

レジスタの桁数 (bit 幅) は, その CPU が最も効率よくデータ処理ができるデータサイズです. この bit 幅は, 時代とともに増加しています.

プログラミング言語 C が開発されたのは 1972 年で, 当時 C が対象にしたコンピュータの CPU のレジスタ幅は, 16bit でした. ちなみに当時は, PC(パーソナルコンピュータ) は存在していませんでした.

その後, PC が現れ (初期の PC は 8bit), C も PC で動作するようになりました. 対象となる CPU のレジスタの bit 幅も, 16bit, 32bit, 64bit となり, C の実装も変化しています.

2.2　規格と実装

教科書にあるように, C は Unix というシステムの開発用として, 仲間内で利用するためのツールとして誕生しました. Unix と C が普及するにつれ, 様々な独自改良が生まれ収拾がつかなくなり, 高級言語の特性が失われそうになりました. (高級言語は, どのような開発環境でも同じようにプログラムが書けなければならない.) そこで, ANSI (American National Standard for Industry) が 1989 年, C の規格を定め, 1990 年, それが国際規格になりました. その後, C の規格は何度かの改訂を経て, 現在に至ります.

規格が定まったとして, C 言語の処理系が「規格を完全に満たす」ように作られているかというと, 必ずしもそうではありません. これは, 古くからある高級言語でも同様です. ただし, 現在では, 規格はほぼ満たされています.

さらに C では,「処理系依存」という形で, 規格にはっきり決められていない部分もあります. これは, 技術の進歩によるコンピュータの機能の向上 (上で言うと, CPU のレジスタの bit 幅が増えることなど) に処理系の機能を合わせやすくするためでもあります.

2.3　char 型

規格上, char 型は,「C のプログラムを記述するのに必要な文字, 記号全体の集合の 1 文字を保持できるデータ量」となっています. C を記述するのに必要な文字は, Ascii コードの文字集合に含まれるので, 規格上は char 型は 7bit 以上の記憶域を持つ変数型となります (6bit だと 64 文字なので少し足りない).

C の char 型の規格をそのまま読むと, 文字コードは, Ascii である必要はありません. 実際 Main Frame(メインフレーム) では, 別の文字コードを利用した C の処理系があります.

この講義で使っている処理系では, char 型は Ascii コードを利用しており, Ascii コードを含むことができ

る 8bit (1 Byte) の整数値を保持する形で実装されており, これは現時点で最も普及している形です. `char` 型を無理に 7bit にする意味もないし, 8bit より大きくする価値もあまりないからです. したがって, `char` 型に代入できる文字は, Ascii コードにある文字だけです. 日本語などの文字扱いは, 大変なので, この講義では述べません. 文字列として UTF-8 文字列は, なんとなく使えることだけは, 知っておいて下さい.

現在では, この実装を逆利用して, char 型を 8bit(1 Byte) の整数を保持するための変数として利用したりします. 例えば, 画像ファイルでは, 画素 (ピクセル, pixel, picture element の略) に対して, 光の 3 原色 RGB (Red, Green, Blue) のそれぞれの値を 1 バイトの大きさでデータを持つ事が多くあります. 画像処理をするソフトでは, この RGB の値を char 型のデータとして処理するように記述する事もあります. 将来的に, この実装が変わることは無いとは思いますが, 本来的には, (処理系の実装に依存した) 変なプログラムの記述です.

もうひとつの char 型の問題点として, 次のような記述が可能である事があります.

```
char moji=65;
```

これは, char 型の変数 moji に Ascii 値が 65 の文字 (大文字の A) を代入して初期化するという, C の仕様としては正しい記述です. しかし, char 型の変数に整数値を代入しているわけですから, 意味的には変です. 「文法的には正しいが, 冷静に考えると意味がおかしい.」というような仕様は, 本来策定すべきではありませんが, (開発現場由来が理由で) C では, このような事がいくつかあります.

教科書 p. 29, プログラム 2.5 を実行して,「文字型は整数値である」ことを実感してください.

2.4 int 型

`int` 型のデータサイズは, もともとは, CPU のレジスタの bit 幅に合わせられてきました. CPU が最も効率よくデータ処理をできるデータサイズだからです.

C が開発されたときには, 16bit(当時の C の処理系の　CPU のレジスタサイズ) で, その後 (1990 年頃?) 32bit が標準的になりました (16bit と 32bit が混在した時期があった).

現在皆さんが使っている処理系でも, `int` のサイズは 32bit(4 Byte) となっているようです. これは, 今の CPU の bit 幅, 64 bit からみると不自然なサイズになっています. これは, 32bit である時代が長く続いたため, その当時のプログラムを変更なく処理するためであろうと思います. 数年後には, `int` 型は 64bit になると思われます. 現在は, 32bit から 64bit へ移行する過渡期だと思われます.

2.4.1 整数の符号の決め方 (2 の補数)

負の整数は, register の桁あふれ (overflow) を利用しています.

簡単のため 4bit で解説します. 4bit の 2 進数 1111 に 0001 を加えると, 10000 となります. しかし, レジスタは 4bit なので最上位の bit は捨てられて 0000 となります. すなわち, 1111 は -1 と同じ振る舞いをします. 正の数と負の数の判定は, 最上位の bit の値で決めます. 最上位の bit が 0 なら正の数, 1 なら負の数とします. 4bit の整数は, 最大値が 0111 (10 進法で 7), 最小値が 1000(10 進法の -8) となるのです. このような負の数の実装を「2 の補数」と言います (1 の補数というのもあったが, 普及しなかった). この計算法で, 符号を反転させるには, 全ての桁の bit 値を入れ替えて (bit 反転という), 1 を加えればよいことになります (例えば, 0111 の符号を変えると, 1001).

一般に, n-bit の符号付き整数では, $2^{n-1}-1$ から -2^{n-1} が数の範囲となります.

整数型には, unsigned, short, long という修飾子がつけられます. 修飾子をつけたときには, int というキーワードを省略できます.

```
unsigned long =unsigned long int
```

それぞれの修飾子の意味は次です.

unsigned = 負の数を考えない.
short = 通常の int より扱える数の範囲が小さくなる.
long = 通常の int より扱える数の範囲が大きくなる.

unsigned では, 整数型の計算は, $\pmod{2^n}$ の計算になります.

short, long を使うことは, 最近では少なくなっています. 下にあるように, 整数のサイズは規格で決まっていないので, 処理系依存のプログラムになるからです. 整数のサイズが問題になるようなプログラムでは, それらの型を別の名前 (例えば, uint32 は, 32bit 符号無し整数の意味) で定義して, 処理系に無関係なプログラムにする工夫をするようになりました (その方法は後の授業で述べます.).

long, short は, 規格では short $\leqq$ int $\leqq$ long を満たすことだけが決められており, 等号が成立する処理系もたくさんあります.

64bit の CPU が出だした頃に, 64bit 整数を利用するために long long という変数型が決められました. 現在では, long long は, 64bit 整数のための変数型として認知されているようです. (これも, 将来は 128bit になるかもしれない).

レポートへのツッコミ: その 1

640320^3 の計算に long 型を利用したレポートがありました. 皆さんの利用している処理系では, long 型は 64bit 整数のようで, 正しい答を計算します. 教科書 p.30, 表 2.5 の long と unsigned long は, 情報処理センターの処理系では, 当てはまりません. しかし, 上で述べたことですが, long = int = 32bit 整数という処理系も, まだたくさんあると思います. なので, 現時点では long long を使うべきです.

2.5 double 型

現在では, 教科書にある IEEE 754 規格に準拠しています. この部分は, ハードウェア (CPU) にあまり関係なくそうなっているようです.

long double 型は, 少し前までは 80bit であることが多かったのですが, 最近は 128bit になっているようです. 浮動小数点計算は, CPU 的に面倒な処理で, そのための特別な回路が用意されていたりしました. それが, 少し前までは 80bit が多かったのですが, 最近 128bit が多くなったからです.

CPU の bit 幅が 8 の倍数になるのは, 昔 IBM が浮動小数点数を 32bit で決めたからである, という説があるのですが, 本当の理由は知りません.

教科書 2.10 節のプログラムを, 教科書を読みながら実行して, nan や inf を体験してください.

レポートへのツッコミ: その 2

教科書のあるように, double 型は仮数部 52bit なので, 640320^3 の値を正確に保持できます. ただし, 次のプログラムは, うまく動きません.

```
#include <stdio.h>

int main()
{
   double a=640320*640320*640320;
   printf("%lu\n", a);
}
```

これは, ある意味, コンパイラのバグです. `double` 型の変数 `a` を宣言して, 同時に 640320^3 を代入しています. コンパイラは, この際に 640320^3 を計算します. 計算した答を代入する方が, 実行時に計算するより動作が速いからです. しかし, 640320 は整数値なので, 整数値のまま 3 乗を実行して整数値のオーバーフローを起こします.

3 情報の欠落

C の基本変数型では, 数値は範囲が有限の範囲に収まるものしか扱えません (その範囲を超えた数値を扱うには, 特別なプログラムを書く必要がある). そのようになった理由は次です.

1. そもそも OS の開発用に作られたので, 開発の初期段階では極端に大きな整数の必要性が無かった.
2. 現時点で, ほとんどの計算において, 十分な範囲と精度があり, 実用的である.
3. 桁数や精度の制限が無い処理系を開発するのは, 大変である.

これが理由で, 正しい計算ができない事があるので注意が必要です.
変数型の範囲以外にもいくつかあるのですが, ここでは, 次を挙げておきます.

異なる型の計算 教科書 p. 26 にあるように, int 型と float 型の計算では, int 型が float 型に変換されて計算されます. 現時点での多くの処理系では, int 型, float 型も 32 ビットのデータになります. ところで, float 型の仮数部は, 23bit しかありません. これは, 24bit 以上の整数型を float 型に変換するときに, 全ての情報を float 型にできない事が起こる事を意味します. その際に情報の欠落がおきます.

レポート問題 (必修ではないが, 実行すれば加点)

連休で時間があるので, 次を実行してみて下さい.

> `http://www.math.u-ryukyu.ac.jp/~suga/gengo/2021/compiler.pdf`
> に従って, 自宅でこの講義を受講する環境を整えよ.

レポート内容は,

- どの環境を実行したか.
- 上の文書で間違っているところ, わかりづらかったところを指摘する.

です.

- 締切: 5 月 9 日 (日) 23:59
- 件名: gengo2021 report 3-1
- 送り先: gengo@math.u-ryukyu.ac.jp