プログラミング発展

移植性とビット演算子とバッファリング

2023年度2Q　火曜日5~7時限(13:45~16:30)
金曜日5~7時限(13:45~16:30)

工学院　情報通信系
尾形わかは，松本隆太郎，
Chu Van Thiem, Saetia Supat
TA:東海林郷志，千脇彰悟

最終更新 7月6日 12:00

移植性について

プログラムの移植性とは…

- プログラムを作成・テストした環境以外でもコンパイルしなおせば問題なく動作する性質を「移植性」と呼ぶ。移植性がある・ない・低い・高いなどと言う
- 他の環境で動作するようにプログラムを改変することを「移植」と呼び、移植の手間が少ないことを「移植性が高い」と呼ぶ

移植性が低いと困る事例

- ちょっと前までアンドロイドスマホのCPUはすべては３２ビットだった。しかし最近になって６４ビットCPUも使われるようになってきた。３２ビットCPUを仮定してプログラムを作っていた業者・個人は移植作業の手間が増えた（あるいは新機種で動かないまま放置された）
- Apple はMacのCPUを2006年ころにPowerPCからインテルに変更した。そして、最近ARM (スマホやスーパーコンピューター富岳のCPU）に変更した（計算機室はインテルMac）
- 次の次の次のページではMacでは正しく動作するがWindowsでは意図通り動作しない事例も紹介する

CPU等の違いによるCプログラムへの影響と移植性の低下要因

- テキストファイルの行末とfopen()のモード（課題9-2,9-3）
- 未定義動作
- 型の大きさが違う
- 定数の大きさが違う、例えば RAND_MAX+1やrand()*10000はウィンドウズでは桁あふれしないが、それ以外の多くの環境で桁あふれしてマイナスの数になる
- 変数を置けるアドレス（メモリ上の番地）の条件が違う
- エンディアンが違う

テキストファイルの改行コードについて

- テキストファイルは行の並びとして構成されている
- 各行の終わりを示す制御文字を改行コードと呼ぶ
- Mac(とLinux)では￥ｎ、Windowsでは￥ｒ￥ｎ、大昔のMac（OSが漢字Talk）は￥ｒであった
- printf等が作られた環境の改行コードは￥ｎだったので、stdio.hで宣言される関数は改行コードを￥ｎのみだと仮定している
- （昔のメモ帳などの）ウィンドウズアプリケーションは行末が￥ｎだけだと行末を認識しないため、fopenのテキストモードでは書き込む￥ｎを￥ｒ￥ｎに変換し、読み込む￥ｒ￥ｎを￥ｎに変換する操作が行われる

fopen()のテキストモードとバイナリモードを適切に使い分ける

- MacOS（とLinux）ではfopen()のテキストモードもバイナリモードもどちらも無変換である（改行コード変換が不要なため）
- 課題9-2, 9-3でテキストモード”r”でfopen()しても何も困らない
- しかしウィンドウズではファイル内の ¥r¥n をプログラムが読み込むと¥n(数値は0x0a)だけに見えて¥r(0x0d)が消える
- これは、無変換でファイルを読まないと困るときにテキストモードでfopen()して移植性が損なわれる事例である
- 読み書きで変換されたら困るときはバイナリモードでオープンする

未定義動作を起こさないようにする

- 符号付き整数桁あふれ（例えばRAND_MAX+1）や、配列の範囲外アクセスなど、実行結果がCPUやコンパイラによってどうにでも変わる処理を、**未定義動作(undefined behavior)**と呼ぶ（再掲）
- 未定義動作がおきるプログラムは、他のCPUやコンパイラを用いたときに**何がおきるかわからない**ので、当然移植性は低い
- 未定義動作の一部を、check1.shや自動採点システムで検出している

型の大きさの違いの例

以下では，Intel = Intel 64-bit CPU, ARM = 32-bit 版ARM (スマホのCPU)

- sizeof(char) = まず間違いなく1バイト
- sizeof(short) = まず間違いなく2バイト
- sizeof(int) = まず間違いなく4バイト
- sizeof(long) = ウィンドウズ以外の64bit CPUでは8バイト、ウィンドウズでは4バイト
- sizeof(void *) = sizeof(long)に等しいことが多いがウィンドウズでは異なる
- sizeof(long long) = まず間違いなく8バイト
- sizeof(float) = まず間違いなく4バイト
- sizeof(double) = まず間違いなく8バイト
- sizeof(long double) = Intel では16バイト、ARMでは8バイト…
- 他CPUにおけるサイズ一覧 https://www.jpcert.or.jp/sc-rules/c-int00-c.html

型の違いへの移植性を高めるために…

- 型の大小関係について

➢char ≦ short ≦ int ≦ long ≦ long long

➢float ≦ double ≦ long double

以外を仮定しない。特にポインタ型と整数型の大小関係は short ≦ void * ≦ long long 程度しか仮定できない（それすらも成り立たない場合があると2018年規格書に記述あり）

- あるいは、次のページに述べる、ビット幅が決まっている整数型を用いる

Stdint.hでtypedefで定義される型

- 1999年に決まった規格C99では stdint.h が新たに作られて、大きさが決まっている整数型が導入された
- intN_t ：丁度Nビットある符号付き整数型で、マイナスの数の表現に２の補数を用いることも決まっている　（int32_t とか int8_tとか）
- uintN_t : 丁度Nビットある符号無し整数型
- intmax_t: その処理系で使える最大の符号付き整数型
- uintmax_t: その処理系で使える最大の符号無し整数型
- intptr_t：void * 以上のビット幅を持つ符号付き整数型(定義されないことあり)
- uintptr_t：intptr_tの符号無し版(定義されないことあり)

変数を置けるアドレス（メモリ上の番地）の条件の違いへの移植性を高める

- ARM CPUの種類によっては int 型変数は4の倍数のアドレスに置かないと実行時エラーが起きる。char型変数・配列が置かれるアドレスは4の倍数とは限らないため、これらをint型としてアクセスすると実行時エラーが起きる（具体例は次ページ）。M1 Macはどうなのかしら？？
- 従って、ある型へのポインタを別の型へのポインタに変換してアクセスすることは原則できないのでやってはいけない
- 例外は標準関数が返す void * ポインタを別の型へのポインタに変換してアクセスする場合と、いかなるポインタも char *, int8_t *, unsigned char *, uint8_t *のいずれかにに変換してバイト単位でアクセスする場合である
- 次ページの例は、どういう結果が出てくるか未定義であり、実行環境によって異なる（可能性がある）。Check1.shと自動採点システムはそのような未定義動作を検出している

変数が置けるアドレスの条件を無視して移植性が低下している例

// 以下のプログラムはインテルCPUでは何かの表示が出ますが、それ以外だと実行時エラーになり何の表示も出てこないことが多い。画像ファイル読み込みで以下のような記述をしないこと望ましい

```
#include <stdio.h>
int main(void)
{
  unsigned i;
  unsigned char array[] = {1,2,3,4,5,6,7,8};
  for (i=0; i<sizeof array - sizeof(unsigned int); i++) {
    unsigned int *pointer = (unsigned *) &array[i]; // インテルCPU以外ではこの行は反則
    printf("*(unsigned int *) &array[%u] = %08x¥n", i, *pointer);
    fflush(stdout);
  }
  return 0;
}
```

変数を配置するアドレスの条件への移植性を高めるためには

- void *以外のポインタを他の型へのポインタに変換してアクセスしない（変換先として許されるのは char *, int8_t *, unsigned char *, uint8_t * のみ）

エンディアンとは何か

int number = 0x010203;

と書くと、各バイトにそれぞれ 0, 1, 2, 3が格納されている。これらのバイトはどういう順番なのか…

char *p = (char *)&number;

としたときに p[0]==0, p[1]==1, p[2]==2, p[3]==3 となっているCPUを**ビッグエンディアン**とよぶ。「京」スーパーコンピューターはビッグエンディアンであった

p[0]==3, p[1]==2, p[2]==1, p[3]==0 となるCPUを**リトルエンディアン** と呼ぶ。インテルはリトルエンディアンである

一般的なエンディアンとファイルへの保存

- 添え字が小さいほう（例えばファイルならより先頭に近いほう）に、（１６進数表現で）大きい桁の数字が書かれる場合にビッグエンディアンと呼ぶ。その逆ならばリトルエンディアンである
- ファイルに数値を読み書きするときに、そのファイルを固定された環境で自分のプログラムだけが読み書きするなら、メモリに書いてある数値の内容をそのままファイルに読み書きしておけばよい
- しかし、数値が(例えば画像形式などの)規格にそってファイルに記録されている場合、そうはいかない

１バイト入出力 fgetc(), fputc()（復習）

- int fgetc(FILE *fp)： fpから1バイト読み0～255またはEOFとして返す。ファイル末尾または読み出しエラーの場合にEOF(たぶん-1）が返される。
- int fputc(int byte, FILE *fp)： ０～２５５の値が入った引数byte（１バイト）をfpに書く。書き込みエラーの場合EOFが返され、書き込めた場合はbyteの値が返される。
- １バイト読み書きはfread()/fwrite()でも可能だが、１バイトの読み書きにはfgetc()/fputc()のほうが使いやすいと感じている（個人の主観である）

ビット演算とエンディアンについて

ビット毎の論理和

- a | b　：　整数型の二つの変数（や定数）のビット毎の論理和をとる
- 2を二進数で表示すると 010 (10とも表示できる) である
- 4を二進数で表示すると 100 である
- 二進数 010 と 100 のビット毎の論理和は 110 だから、十進数で表示すると 2 | 4 == 6 である
- 6を二進数で表示すると 110 である
- 二進数 010 と 110 のビット毎の論理和は 110 だから、十進数で表示すると 2 | 6 == 6 である
- **ここで、a, bの二進数表示の1の桁に重なりが無いなら、 a | b == a+b である。これを課題10-1, 10-2で用いる**

ビット毎の論理積

- a & b ： 整数型の二つの変数（や定数）のビット毎の論理積をとる
- 二進数で010 & 100 == 000だから十進数で2 & 4 == 0,
- 二進数で010 & 110 == 010だから十進数で2 & 6 == 2。
- 0xf の二進数表現は 1111 であるから、0xf と論理積をとれば１６進数のある桁を抽出できる（例： 0x1234 & 0xf000 == 0x1000, 0x1234 & 0x00ff == 0x0034）
- 変数の型が符号なし unsigned でも符号あり signed でも動作は変わらない
- 高校教科書の二進数の説明をT2Schola第１回に置きました
- 二進数がわからない人には**TAが親切にサポート**します

ビットシフト

- a >> n　：整数aの二進数表現を右にnビットシフトする。aが非負のとき a / (2のn乗) に等しい。 aが**負の値の場合には何が起きるか未定義**。
- 例： 6 >> 0 == 6, 6 >> 1 == 3, 6 >> 2 == 1, 6 >> 3 == 0, …
- a << n　：整数aの二進数表現を左にnビットシフトする。結果が桁あふれしなかったとき a * (2のn乗) に等しい。桁あふれしない限り a が負でも問題ない
- ビット毎論理和・論理積より高い優先度を持つ、例えば bytes[0] | bytes[1] << 8 は bytes[0] | (bytes[1] << 8) を表す

４バイトの数値の読み出し

ファイル “little.bin” の 11, 12,13,14バイト目にリトルエンディアンで符号なし数値が書いてあるとする。これを読んでunsigned int型変数に格納するにはどうすればよいか？

ファイル内に（リトルエンディアンで）書いてある整数の読み書きは次回の課題で必要になる予定

数値のファイルへの記録方法がリトルエンディアンでの読み出し例

FILE *fp = fopen(“little.bin”, “rb”); //**バイナリモード**

unsigned int i, number; long long bytes[4];

fseek(fp, 10, SEEK_SET); // 10バイトスキップする

for (i = 0; i<4; i++) {

  bytes[i] = fgetc(fp);

  if (bytes[i] == EOF) { fprintf(stderr, “read error!”); exit(1); }

}

number = bytes[0] | bytes[1] << 8 | bytes[2] << 16 | bytes[3] << 24;

でファイル上の数値を変数numberに読み出せる

ダメな例

```
number = fgetc(fp) | fgetc(fp) << 8 |  fgetc(fp) << 16 | fgetc(fp) << 24;
```

ダメな理由１：代入等号の右辺にある fgetc() の呼び出しの順序はまったく未定義であり、左から呼び出される保証がないため

ダメな理由２：fgetc(fp) の値が１２８以上なら、 fgetc(fp)<<24 は符号付きintの最大値を超えて**桁あふれし**、値がどうなるか全くわからなくなる。同様に配列 bytes をunsigned int型にするのも間違い。unsigined int なら桁あふれしないが、EOF (-1) と==にならなくなり読み出しファイル終端や読み出しエラーを認識できなくなる

リトルエンディアンでの数値書き込み

```
unsigned int number = 0x04030201;
fputc(number & 0xff, fp); // 0x01 が書かれる
fputc(number >> 8 & 0xff, fp); // 0x02 が書かれる
fputc(number >> 16 & 0xff, fp); // 0x03 が書かれる
fputc(number >> 24 & 0xff, fp); // 0x04 が書かれる
```

注意：変数numberを符号付き整数（ただのint等)で宣言しnumberが負の値であるとき、右シフトの結果が未定義になるので、そういう書き方はさけるべき

バッファリングについて

(f)printf() で結果がファイルに反映されない

- 研究室に所属すると、長い時間がかかるシミュレーションや数値計算をプログラムし、途中結果を(f)printf()で随時書き出し、ファイルの内容をときどき確認することがある
- そのときに、何も工夫せずに (f)printf() するとプログラムが(f)printf()を実行したのに内容がファイルに反映されないことがある
- Eclipseでprintf()した文字列が画面に表示されない（ことがある）
- その原因と対応策を説明する

バッファリングとは

- ハードディスクなどの遅いデバイスやシステムに対する読み書きで、プログラムの動作速度が低下することを避けるためのテクニックである
- 書き込みが遅い処理が求められるときに、**バッファ**と呼ばれるメモリに書きたいデータを蓄積し、遅い書き込み処理を一括（場合によっては並行）して行うこと、ならびに
- 読み込みが遅い処理が求められるときに、読みたいデータのさらに先にあるデータまでまとめて読んで**バッファ**に格納する２つの処理からなる

OSが管理するバッファ

- メモ帳やエクセルなどのプログラムがファイルへの書き込みをOSに依頼したときに、OSはハードディスクやUSBメモリなどの処理が遅い外部記憶装置への書き込みを行わずに、 **OSのバッファ**に内容をコピーする.
- OSのバッファに書き込まれた内容は、他のプログラムがそのファイルを読みだしたときにも見える.
- OSは、OSのバッファの空きが少なくなったときなどに、その内容を外部記憶装置に書き出している。（この処理はメモ帳などと並行して実行され、メモ帳などの動作を遅くしない）
- そのため、外部記憶装置を突然引き抜いたり、コンセントを抜いてOSを突然終了すると、メモ帳などが保存した内容が失われることがある

stdio.hが管理するバッファ

- ファイルへの読み書きをOSに指示する操作はメモリアクセスと比較すると若干遅い（その遅さを体験することが今回に課題に含まれる）
- その遅さをさけるために、fprintf(),fputc(),fwrite()などでは，逐次OSへのファイル書き込み指示を行うことはせず、**stdio.hのバッファ**に書き込む内容をコピーしておく
- stdio.h のバッファは、一杯になるか fflush() されたときに，write()を用いてOSに書き込み指示される。write()の実行時間は書き込みが1バイトであってもBUFSIZバイトであってもほぼ変わらないため、write()呼び出しをまとめて回数を減らすと動作が速くなる。
- fread()でファイルから読み込むときも 必要なデータを含むBUFSIZバイト分をまとめて**stdio.hのバッファ**に読み込んでおくことで，動作を速くできる.
- stdio.h のバッファの内容がwrite()によりOSに渡される前は、fprintfなどで内容を書き換えたファイルを他のプログラムから見ても，変更内容が**見えない**

stdio.h のバッファとOSのバッファの図

自作Cプログラム

オレンジ色は計算機メモリ内にある

高速な読み書き。fprintf(),fputc(),fwrite()はstdio.hのバッファに内容を書く

stdio.h のバッファ

若干遅い読み書き。fflush() でstdio.hのバッファの内容がOSのバッファにコピーされる

OSのバッファ＝一般プログラムから見えるファイル

相当遅い読み書き。sync(),fsync(),FlushFileBuffers()等でOSのバッファの内容を外部記憶装置へ書き出しできる。OSのバッファを書き出す前に外部記憶装置を引き抜いたり、（コンセント引き抜き等で）OSを突然終了すると、外部記憶装置の内容は滅茶苦茶になりファイルが消滅したり一部分アクセス不能になることがある

外部記憶装置

fflush() によるバッファ内容の書き出し

- fflush(fp) によりファイルポインタ fp に付属するstdio.hのバッファがため込んでいる内容を、(MacOSの)write()を呼び出すことで、実際にファイルに反映させることができる
- (f)printf() の内容が反映されてほしいなら fflush() を呼ぶべきで、厳密に言えば printf() の内容が表示されて欲しい場合 fflush(stdout) を呼ぶべきではある（しかし呼ばなくても問題が生じない環境が多い…）
- 詳しくは述べないが、標準関数の setvbuf() を用いて指定したファイルポインタについてstdio.hのバッファリングを完全に禁止することもできる

バッファリングにより生じる未定義動作とその防止法

読み出しと書き込みの切り替えとfflush

- “r+”, “w+”, “a+” で fopen() したファイルポインタfpについて
- ✓ 読み出したあとに初めて書き込むとき
- ✓ 書き込んだあとに初めて読み出すとき
- に必ず fflush()またはfseek() （またはfsetpos()）を実行しないと何がおきるかわからない(未定義動作)。バッファの書き出しや読み書きする位置を動かす必要が無ければ fseek(fp, 0, SEEK_CUR) を実行すればよい。fflush()またはfseek()を実行しないと移植性が低くなる
- 出展 https://www.jpcert.or.jp/sc-rules/c-fio39-c.html

同一ファイルを２回fopen()すると未定義動作

- 同一のファイルをfclose()せずに２回fopen()すると何が起きるかわからない
- ２回目のfopen()に失敗するかもしれない
- ファイルに書き込んだ内容が反映されないかも知れない
- ファイルの同一の場所を同時に読んでも内容が異なるかも知れない
- これら怪現象が起こりえる理由の一つは、fopen()するごとに別々のバッファが作られ、２回のfopen()で得られた２つのファイルポインタに対する操作で、２つのバッファが矛盾した内容（異なる内容）になるからである

Ex 10-1（ビット演算でエンディアン）

- ファイルlittle2.bin には、第1,2,3バイトにリトルエンディアンで符号なし（unsigned int)の数値Aが書かれている。同様に第4,5,6バイトにリトルエンディアンで符号なし（unsigned int)の数値Bが書かれている。
- 以下を行うプログラムを作成しなさい。

1. little2.bin からAとBを読み込み、A×Bを計算し、計算結果を**リトルエンディアンで**little2.binの第7,8,9,10,11,12バイトに書く。
2. Ex9-3で作成した(あるいは解答例で示された) file_dump関数を用いて、変更後のlittle2.binの内容を表示する。

- 提出ファイル名 ex10_1.c

注意：little2.binの第6バイト目以前および第13バイト目以降を書き換えないこと。

Ex 10-2（ビット演算でエンディアン）

- ファイルbig2.bin には、第1,2,3バイトにビッグエンディアンで符号なし（unsigned int)の数値Aが書かれている。同様に第4,5,6バイトにビッグエンディアンで符号なし（unsigned int)の数値Bが書かれている。
- 以下を行うプログラムを作成しなさい。

1. big2.bin からAとBを読み込み、A×Bを計算し、計算結果を**ビッグエンディアンで**big2.binの第7,8,9,10,11,12バイトに書く。
2. Ex9-3で作成した(あるいは解答例で示された) file_dump関数を用いて、変更後のbig2.binの内容を表示する。

- 提出ファイル名 ex10_2.c

注意：big2.binの第6バイト目以前および第13バイト目以降を書き換えないこと。

注意事項

- シフトとビット毎演算を使ってほしいため、足し算＋、引き算ー、掛け算 *、割り算 / 、剰余 % 演算子の**使用をA×Bの計算を除き禁止**する
- 課題のリトルエンディアンとビッグエンディアンは、ファイルに整数を記録する形式について指示している。それは、計算機内で変数がメモリに格納されるときのエンディアンとはまったく関係がない
- 課題のlittle2.bin, big2.binは~matsumoto.r.aa/bin/biglittle.shを実行すると以前のrandom-matrix.shと同様に毎回異なる内容で生成される。自宅実習している場合自分でファイルを作成して下さい
- 同一ファイルをfclose()せずに２回fopen()しないこと
- 読み出しと書き込みの間にはfflush(),fseek(),fclose()のどれかを挟むこと

計算結果の自己確認

- 課題10-1,10-2について正しく結果をファイルに書き込めたのか自分で判断することが若干難しい
- ~matsumoto.r.aa/bin/check-ex10-12 を用いてlittle2.bin および big2.bin に書き込んだ積が正しいか否か確認できる
- 上記の確認プログラムを用いるなら、~matsumoto.r.aa/bin/check1.sh を用いて生成したa.outを使ってlittle2.binまたはbig2.binに積を書き込んで、その積をcheck-ex10-12で確認すると、自動採点システムに近い厳しさでプログラムの間違いを洗い出せる
- biglittle.shで色々な内容のlittle2.bin, big2.binを作成し上記を確認するとよい。特定の内容だけで発生する良く起きる誤りがあるため

自己確認できないとクビになる

- 学校（研究室所属以前）では、やったことの正否の確認は他人任せである
- 会社で依頼された業務についてやったことの正否を依頼者に丸投げすると、依頼者に確認の手間を押し付けることになる
- やったことがほぼ常に正しいなら依頼者は確認を省略できて手間が増えないが、間違うことがよくあると手間が増えて「私の手間を増やすならあなたには頼まず他の人に頼む」ということになる
- 何らかの業務で、依頼者に確認の手間を押し付けず依頼を完遂できないと、その会社で頼まれる業務が無くなる
- （そうなってもクビにならない会社に潜り込む戦略もある…）
- 大学の研究室は会社ではないからクビにはならないが…？
- やったことの正否を自分で検算できることが重要

Ex 10-3（バッファリングによる高速化を体験する）

- ファイルtest.binに０から２５５のランダムな数値をfputc() で十万回書き出すときに、バッファリングによる速度向上の効果を調べる
- fputc() の直後にfflush() を実行する場合としない場合で十万回の書き出しに掛かる時間をそれぞれ**表示する**プログラムを、**単一**のCプログラムとして作成せよ
- 調べた結果（計測した時間）をCプログラムの冒頭にコメントとして記入し提出せよ。提出物 ex10_3.c

Ex 10-4
（エンディアン・最終課題への伏線）

以下を行うプログラムを作成しなさい。ただし、step 4で表示された値が、入力した十進数と同じになるように、step 3で代入すること。

1. 共用体の型
   typedef union {
   　unsigned long long ull;
   　unsigned char uc[8];
   } ulonglong_uchar;
   を定義し、適当な名前で変数を宣言する
2. char string[128]; に、scanf %sを用いて十進数を読み込む。
3. ０～２５５の数値を上記共用体変数の uc[0]～uc[7] に代入する。（ullには代入を行わない）
4. 共用体変数のullをprintf %lluで表示する。

（続く）

Ex 10-4
（エンディアン・最終課題への伏線）

- 提出物：ex10_4.c
- 提出する前に、入力として0, 1, 123456789012345 の３通りについて、正しく動作することを確認すること。
- 入力十進数がunsigned long long最大値を超えるときには異常動作して構わない

補足

- CPUがリトルエンディアンのとき
  ull == uc[0]+uc[1]*256ULL+uc[2]*256ULL*256ULL+…
  が常に成り立つ （ULLは数値がunsigned long long型であることを示す）
- 最終課題で使うhuge_int型内の配列numとそれが表現する整数の関係は、ucとullの関係と同じになる

- 時間があまった場合、次回の講義資料がT2Scholaにあるので（？）予習して次回の課題に着手して下さい
- 12回目はMac（またはLinux）が必要なので、自宅実習でWindowsしかない人は計算機室に**来られるなら**来たほうがいいです。WSL (Windows Subsystem for Linux)で12回目の課題をできるかどうかは確認していません。
- コロナ等で登校できない人のために、12回目以降を自宅実習するための情報（手順）は12回目以降講義資料に含めてあります

提出ファイル一覧

- ex10_1.c
- ex10_2.c
- ex10_3.c
- ex10_4.c