基礎プログラミング+演習 ＃15 – チームによる開発(総合実習)

久野 靖 (電気通信大学)

2017.12.11

今回の内容は「総合実習」であり、2〜3 名のグループで協力して「動画を生成する整った構造のプログラム」を開発して頂きます。したがって課題は「B 課題」のみです。今回の目標は次のことがらです。

- チームでソフトウェアを開発する際に注意すべきことを知る。
- C 言語の機能を活用して分担してプログラムを開発する。

1 前回演習問題の解説

1.1 演習 1 — 色の構造体

色の構造体の演習は簡単だと思います (HSB 変換は面倒ですが)。実数計算する場合は、最後に整数に戻すため `int` へのキャストが必要なことに注意してください。また、`randomcolor()` で乗数が 256 でいいのかなと思うでしょうけれど、乱数が 1「未満」なので掛け算して切り捨てたら 255 以下になるわけです。

```
struct color brighter(struct color c) {
  struct color white = { 255, 255, 255 };
  return mixcolor(c, white);
}
struct color darker(struct color c) {
  struct color black = { 0, 0, 0 };
  return mixcolor(c, black);
}
struct color linearmix(struct color c, struct color d, double p) {
  double q = 1.0 - p;
  struct color c1 = {
    (int)(c.r*p+d.r*q), (int)(c.g*p+d.g*q), (int)(c.b*p+d.b*q) };
  return c1;
}
struct color randomcolor(void) {
  struct color c1 = {
    (int)(256*drand48()), (int)(256*drand48()), (int)(256*drand48()) };
  return c1;
}
```

1.2 演習 2 — 線形探索の表

演習 2 は線形探索の表に機能を追加するものでした。ここでは削除と全部表示の 2 つを示します。まず `tbl.h` にこれらの関数の宣言を追加します。

```
bool tbl_delete(char *k);
void tbl_show(void);
```

tbl_delete が bool を返すのは、見つかって削除したかそのキーの項目は無かったかの区別を返すというつもりです。そして削除ですが、これまでと同様に探して見つかったときは削除します。削除するときは表の最後の項目をその位置にコピーしてきて表のサイズを 1 つ減らせばよいですが、サイズが 1 のときはコピーしてくるものがないのでコピーしません。あと、malloc で割り当てた文字列領域は不要になったら free で返却するべきなのでそうしています。

```
bool tbl_delete(char *k) {
  for(int i = 0; i < tblsize; ++i) {
    if(strcmp(tbl[i].key, k) == 0) {
      free(tbl[i].key);
      if(tblsize > 1) { tbl[i] = tbl[tblsize-1]; }
      --tblsize;  return true;  // found and deleted
    }
  }
  return false; // not found
}
```

そして全部表示はむしろ簡単ですね。

```
void tbl_show(void) {
  for(int i = 0; i < tblsize; ++i) {
    printf("%s: %d\n", tbl[i].key, tbl[i].val);
  }
}
```

これらをテストする main の方も変更しました。-2 を入力したとき削除、そして終わるときに全部表示します。

```
int main(void) {
  char buf[MAXBUF];
  int val;
  while(true) {
    printf("key (empty for quit)> ");
    if(fgets(buf, MAXBUF, stdin) == NULL) { return 1; }
    chopnl(buf, MAXBUF);
    if(strlen(buf) == 0) { break; }
    printf("val (-1 for query, -2 for del)> ");
    scanf("%d", &val);
    if(val == -1) {
      printf("tbl[%s] == %d\n", buf, tbl_get(buf));
    } else if(val == -2) {
      tbl_delete(buf);
    } else {
      tbl_put(buf, val);
    }
    if(fgets(buf, MAXBUF, stdin) == NULL) { return  1; }
  }
```

```
  tbl_show(); return 0;
}
```

では実行例です。

```
% ./a.out
key (empty for quit)> kuno
val (-1 for query, -2 for del)> 5
key (empty for quit)> nakano
val (-1 for query, -2 for del)> 10
key (empty for quit)> sasaki
val (-1 for query, -2 for del)> 15
key (empty for quit)> kuno
val (-1 for query, -2 for del)> -2
key (empty for quit)>
sasaki: 15
nakano: 10
%
```

削除するときに最後の要素をコピーしてくるので、最後に全表示すると `sasaki` の方が前になっているのがわかります。

2 チームによるソフトウェア開発

2.1 ソフトウェア開発の難しさ

ここまで様々なプログラムについて扱ってきましたが、世の中では「ソフトウェア」という用語の方が多く使われます。一般に**ソフトウェア** (software) とは、プログラムとそれを動かすのに必要なデータ等を合わせたものを言います。

世の中のソフトウェアは数十万行以上のプログラムコードを含むものも珍しくなく、そのようなものは一人では開発できないので、チームで開発することになります (もちろん、もっと小さい規模でも必要があればチーム開発が行なわれます)。

一人でただプログラムを作るのと比較して、チームによるソフトウェア開発では次のような追加の作業が必要です。

- 仕様策定 — どのようなソフトウェアを作るかを決める。
- 設計 — プログラムやデータの構成や形を決める。
- 開発管理 — 分担やスケジュールを決めて進捗を管理しながら開発する。
- テスト・デバッグ — 作成したソフトが仕様通り動くか検査し不具合があれば修正する。
- 運用・保守 — ソフトを動かしつつ改訂や不具合修正を行なう。
- 文書化・記録 — 上記すべての作業について記録して残す。

実際には 1 人であっても、ソフトウェアを「きちんと」作るのであればこれらの作業はすべて必要なことです。

さらに、ソフトウェアが大規模で関係する人数が多くなると、これらの作業内容を複数人で打ち合せて調整する手間が非常に大きくなります。このため、ただプログラムを書くのであれば 1 日に何百行も書けるようなソフトウェア開発者でも、仕事としてプロジェクトでソフトウェア開発を行なう場合は、平均すると 1 人あたり 1 日数行程度しか書いていない計算になると言われています。

2.2　ソフトウェア工学とソフトウェア開発プロセス

過去においても現在においても、実際にソフトウェア開発をおこなうと、様々なトラブルが発生しています。典型的なものをいくつか挙げます。

- どのようなソフトウェアを作るのかの仕様策定がいつまでも終わらずに開発に入れない。
- 仕様策定して開発したものができあがってみると発注側からこれでは使えないと言われる。
- 仕様を決定して開始したはずなのに途中で変更が次々に現れてつぎはぎだらけのソフトウェアになる。
- プログラムの品質が悪くバグだらけでいつまでも開発が終わらない。
- プログラムが完成して運用に入るが重大なバグが残っていてトラブルが発生する。

このような問題の多くは、ソフトウェア開発が非常に緻密な作業であり 1 箇所の間違いでも重大な障害につながる可能性があるということに由来しています。また、最後にソフトができあがって動かしてみないとどのようなものか分からないから、という面もあります。

このような多くの問題を何とかしようとする研究の分野を**ソフトウェア工学** (software engineering) と呼びます。その名前からすると「プログラムを作ることの研究」みたいですが、実際には発注者にきちんと確認するとか進捗を管理するとか、人間にまつわる事柄の多い分野です。

その中に、どのような流れでソフトウェアを開発するか、という分野があり、そこでは具体的な開発の進め方のことを**ソフトウェア開発プロセス** (software development process) と呼んでいます。

過去においては「分析→設計→正属→テスト→保守」のように段階を経てソフトウェアを開発していくプロセス (ウォーターフォール型) が主流でしたが、このやり方だと「分析・設計の結果が後になって違っていると分かって手戻りになる」問題が大きいと分かってきました。

そのため今日では、作成する機能の優先順位つきリストを作り、優先度の高い機能をとりあえず実現して動かし、動作を確認してから次の機能を追加することを反復していくプロセス (アジャイル型) が広まって来ています。

皆様がソフトウェアを開発するときも、後者のやり方を強く勧めます。最初に大きな完成形を描いてしまうと、そこまでの道のりが遠くて挫折しやすいですし、完成するまで動かないので組み立てて動かそうとしたときにはコードが大量になっていて、どこが間違っているか分からず困る、ということになるからです。

本資料でもいくつか、やや大きなプログラムの例がありましたが、いずれも「まず小さく作って動かす」「動いたら徐々に機能を増やして行く」というやり方で説明されています。この方法だと、機能を追加したときにトラブルが起きたら、その追加したあたりを見ればよいと分かっているので、はるかに問題を解決しやすいのです。

2.3　C 言語の機能と共同作業

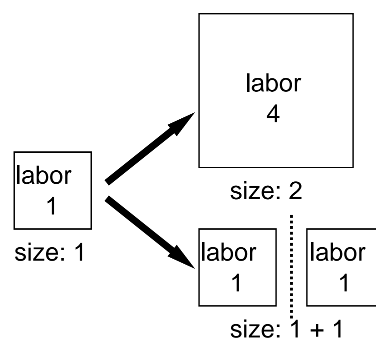

図 1: プログラムを独立した部分に分ける必要性

大きなプログラムを設計するときに重要な指針として、「部分ごとの独立性を高める」ということがあります。一般に、サイズ S のプログラムに対して、その 2 倍、$2S$ のサイズのプログラムは作ったり理解したりする労力が 4 倍くらいかかる、という面があります。それは、プログラムのどの箇所でもほかの様々な箇所と相互作用する可能性があるから、サイズが 2 倍に掛け算してそれぞれの部分の相互作用が 2 倍になる、と考えればよいでしょう (図 1 上)。

ここでプログラムの設計を見直し、そのプログラムを互いにほとんど関係しない (数箇所で呼び出すだけの)2 つの部分に分けることができたら、$S + S$ でもとの 2 倍の労力で開発できます (図 1 下)。すなわち、プログラムの部分どうしができるだけ関係しないようにすることが大切なのです。

C 言語でそのような分離を実現するには、「特定用途の機能の集まり」を 1 つのファイルに入れる形で行なうのが定石です。既に見てきたように、C 言語ではグローバル変数に `static` を指定することで、その変数がファイル内だけで参照できるものになります。それぞれのファイルをそのようにすることで、個々のファイル内のコードは他のファイルからほとんど独立したものとできます。

あるファイル内から別のファイルで実現されている機能を利用するには、もちろんそのファイルの関数を呼び出します。そのためにはプロトタイプ宣言が必要ですが、それはファイルごとに対応するヘッダファイルに用意すればよいのです。そして各ファイルにおいて、よそのファイルから呼び出されることを想定しない関数は、やはり `static` を指定してよそから参照できなくしておきます (図 2)。

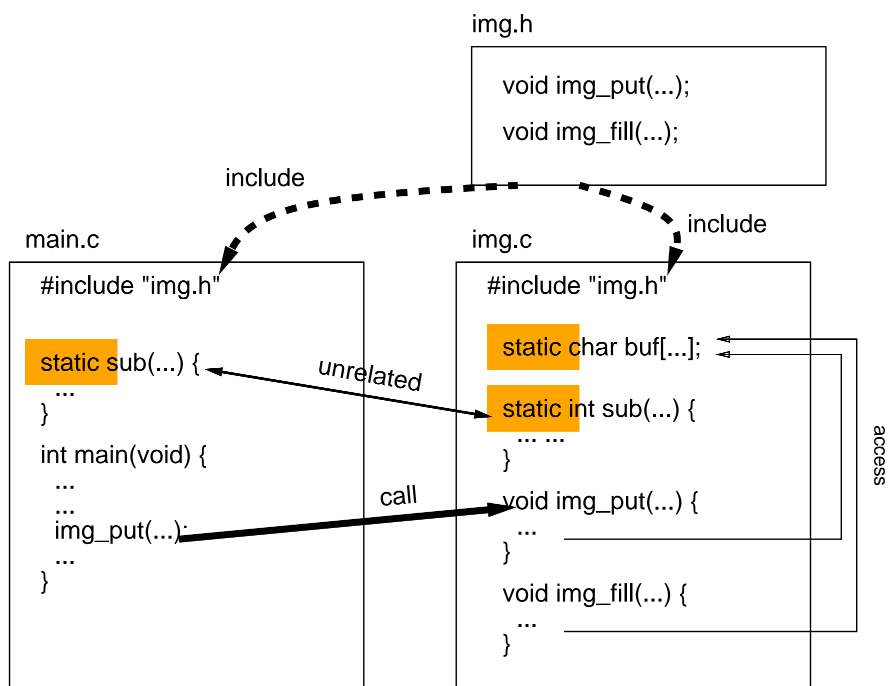

図 2: C 言語でファイルを分けて扱う

一般に、ひとまとまりの機能に対して、その機能を外部から (たとえばよそのファイルから) 呼び出すときに使う関数の集まりのことを **API**(application programming interface) と呼びます。プログラムをうまくいくつかの機能に分け、それぞれの機能ごとにうまく設計された API を定義し、それらを呼ぶことでプログラム全体が動作する、というのが整ったプログラムの形だと考えてよいでしょう。

3 動画ファイルの API を作る

3.1 API の設計

それでは今回の例題として、Ruby でも扱った PPM 画像ファイルの出力を取り上げます。ただし今回は、動画を生成するために多数の PPM 画像ファイルを出力することを想定します。そこで、API

を定義するヘッダファイル img.h を見てみましょう。

```
#define WIDTH 300
#define HEIGHT 200
struct color { unsigned char r, g, b; };
void img_clear(void);
void img_write(void);
void img_putpixel(struct color c, int x, int y);
void img_fillcircle(struct color c, double x, double y, double r);
```

まず画像の幅と高さはここで定義しています。次に色については前にやったように構造体 struct color で定義しています。そして残りの関数は次のようにします。

- img_clear — 画像を真っ白に初期化する。
- img_write — 現在の画像を PPM 形式でファイルに書き出す。ファイル名は img*dddd*.ppm に固定で、*dddd* のところは 0001, 0002, ... と書くごとに番号が進んで行くものとする。
- img_putpixel — 指定した色で指定した (x, y) 位置に点を打つ。
- img_fillcircle — 指定した色で指定した (x, y) を中心とし半径 r の円を塗りつぶす。

ここでなぜ円の方だけ座標や半径が実数なのかと思ったかも知れませんが、アニメーションをやるということは座標や半径を連続的に変化させたいので実数が便利なのです。putpixel の方は直接呼ぶことはあまりなさそうなので整数のままにしました。

3.2 API の実装

では上で設計した API の実装を見ます。画像データを入れる配列 buf は構造体の 2 次元配列ではなく、文字の 3 次元配列にしました。その理由は、C 言語では 3 バイトの大きさの構造体を配列にするとアクセスを高速にするため 1 バイトの「詰めもの」をして 4 バイトにするという機能がくっついていて、そうすると画像としてそのまま書き出せないからです (難しいと思いますがそういうものだと思ってください)。文字だけの配列ならこのようなことは起きません。変数 filecnt はファイルにつける連番、fname はファイル名生成用の領域です。

クリアは簡単で、すべてのピクセルの RGB 値をすべて 255(真っ白) にします。書くときは、まずファイル名を fname に生成し、次に fopen でその名前のファイルを書き出しモードで準備します。ファイル生成が失敗すると NULL が返されるのでそのときはエラーメッセージを出して終わります。OK なら、そのファイルにまず PPM 画像のヘッダ部分「P6、幅 高さ、255」を書き、続いて buf 全体をいっきに出力します。fwrite は指定したポインタ値の場所から指定したバイト数のかたまりを N 個ぶん (今回は 1 個を指定) 書き出します。

```
#include <stdio.h>
#include <stdlib.h>
#include "img.h"
static unsigned char buf[HEIGHT][WIDTH][3];
static int filecnt = 0;
static char fname[100];

void img_clear(void) {
  for(int j = 0; j < HEIGHT; ++j) {
    for(int i = 0; i < WIDTH; ++i) {
      buf[j][i][0] = buf[j][i][1] = buf[j][i][2] = 255;
    }
```

```
  }
}
void img_write(void) {
  sprintf(fname, "img%04d.ppm", ++filecnt);
  FILE *f = fopen(fname, "wb");
  if(f == NULL) { fprintf(stderr, "can't open %s\n", fname); exit(1); }
  fprintf(f, "P6\n%d %d\n255\n", WIDTH, HEIGHT);
  fwrite(buf, sizeof(buf), 1, f);
  fclose(f);
}
void img_putpixel(struct color c, int x, int y) {
  if(x < 0 || x >= WIDTH || y < 0 || y >= HEIGHT) { return; }
  buf[HEIGHT-y-1][x][0] = c.r;
  buf[HEIGHT-y-1][x][1] = c.g;
  buf[HEIGHT-y-1][x][2] = c.b;
}

void img_fillcircle(struct color c, double x, double y, double r) {
  int imin = (int)(x - r - 1), imax = (int)(x + r + 1);
  int jmin = (int)(y - r - 1), jmax = (int)(y + r + 1);
  for(int j = jmin; j <= jmax; ++j) {
    for(int i = imin; i <= imax; ++i) {
      if((x-i)*(x-i) + (y-j)*(y-j) <= r*r) { img_putpixel(c, i, j); }
    }
  }
}
```

putpixel は指定したピクセル位置にレコードから RGB 値をコピーしますが、ただし画像の上の方が Y 軸で正の向きにしたいので、0 が画像の一番下の行になるように引き算を使っています。

fillcircle は画像上の円が含まれる範囲をまず整数で計算し、その範囲すべての点について、円の中に入っていたら putpixel で点を打ちます。

3.3 動画を作り出す

動画の原理は「少しずつ違う画像を次々に表示すると動いて見える」ということはご存じですよね。では、動画を作り出してみましょう。非常に簡単なプログラムです。まず色を 2 つ用意し、1 番目の色で 20 フレームぶん、だんだん位置を横に動かしながら円を描きます。そのあとさらに 20 フレームぶん、こんどは 2 番目の色でだんだん上の位置に動きながら、半径が小さくなるように円を動かします (図 3)。

```
// animate1 --- create animation using img lib.
#include "img.h"

int main(void) {
  struct color c1 = { 30, 255, 0 };
  struct color c2 = { 255, 0, 0 };
  for(int i = 0; i < 20; ++i) {
    img_clear(); img_fillcircle(c1, 20+i*8, 100, 20); img_write();
  }
```

```
  for(int i = 0; i < 20; ++i) {
    img_clear(); img_fillcircle(c2, 180, 100+i*5, 20-i); img_write();
  }
}
```

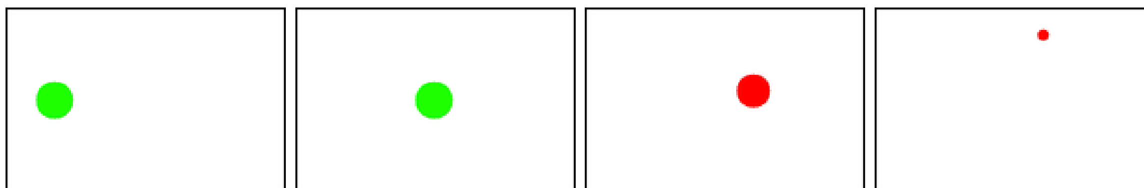

図 3: 動画のコマの例

実際にこれを動いて見えるようにするためには、**アニメーション GIF**(animation GIF) 形式に変換してください。次のようにすればよいのです。

```
% gcc animate1.c img.c -std=c99  ←普通にコンパイル
% ./a.out                        ←実行
% animate -set delay=5 img*.ppm  ←アニメーション表示
% convert -set delay=5 img*.ppm out.gif  ←アニメ GIF に変換
```

`convert` で生成した GIF ファイルはブラウザで開いてください (普通のブラウザにはアニメーション GIF 再生機能がついています)。`animate` というコマンドで直接アニメーション表示もできます。テスト中はこちらが便利ですが、後で色々な人に見せるときはアニメーション GIF の方がよいでしょう。

3.4 課題のためのヒント

課題は動画を生成するプログラムなので、もちろん上の例題を利用して頂いて構いません (独自に設計したければそうされても構いません)。上の例題を利用するとして、「整った構造」についてはどのように考えたらいいでしょうか。

まず、例題ではほとんど円しか描けないので、ほかの図形を追加したいですね。もちろん `main` の方で直接点を打って図形を描くことはできますが、プログラムを複数の部分にきれいに分けるという点では、`img.c` の中に三角形とか長方形などを描く関数を追加する方が整っていると思われます。

さらに、複雑なシーンを持つ絵であれば単純な図形ではなく「家」とか「車」とか図形の組み合わさった形が現れると思われます。このとき、毎回 `main` から三角形や長方形や円の関数を呼び出すより、「家」や「車」という関数があってそれを呼び出す方がよさそうですね。

その「家」「車」という関数はどこに入れるのがいいでしょうか。`main` と一緒に入れるとか、`img.c` に入れてしまうとか、別のファイル `parts.c` を作ってそこに入れるとか、複数の選択肢があると思います (これはどれが正解ということはないですが、どのようにするかはきちんと考えて決めてレポートに書いて頂きたいです)。

次に動画なのでどのように動かすかも問題です。例題では直接フレームという単位でフレームごとにどれだけ動かす/小さくするなどを扱っていましたが、複雑な絵や複雑な動きだとごちゃごちゃになります。

そもそも動くということは、時間とともに位置や大きさが変化するということですから、時間に関する関数 $(x, y) = f(t)$ のようなものを定義して使うと「整って」いるかも知れません (t は秒数で与えたいですが、たとえば 20 フレームで 1 秒とか適当に決めればいいと思われます)。

または、複数の場面から構成される演劇のような動画も考えると、「この円は時刻 $t1$ から $t2$ の間存在し、その間に $(x1, y1)$ から $(x2, y2)$ までなめらかに移動し、かつ大きさは $r1$ から $r2$ に変化する」のような指定ができることが望ましいかも知れません。そうすると、そのようなものは当然 `main` で直接やることではなく、また `img.c` でやることでもなく、その中間にある `anim.c` のようなファイル

が `img.c` を呼び出しつつ「動く円」「動く長方形」など必要なものを提供し、`main` はこれを利用して動画を組み立てて行く、というふうになるかも知れません。

なお、これらはすべてヒントなので、どのくらい何を工夫するかはそれぞれのチームで相談して決めてください。元の例題の構造のままで整っているということでもいっこうに構いません。

また、さまざまなファイルの分け方の話もしましたが、課題をチームでやる場合には、ファイルごとに担当するとか、1 人が設計をして他方が作るとか、誰かは記録だけするとか、これもそれぞれのチームにお任せします。ただし、何も仕事をしない人が出ることは避けてください。

報告課題 **15A**

今回は総合実習のため当日は「報告課題」(時間中にやったことの報告) です (プログラムの提出は不要)。簡単にまとめてください。

- Q1. どのような分担で課題プログラムを構成する計画ですか。
- Q2. 複数で分担して 1 つのプログラムを作成することをどう思いますか。
- Q3. リフレクション (今回の課題で分かったこと)・感想・要望をどうぞ。

総合実習課題 **15B**

B 課題は必ず 2〜3 名のグループで実施し、かつ、ペアプログラミングではなく「各自が自分の担当を書く」形にしてください (どうしてもメンバーが見つからない時は担当教員に相談のこと)。課題は次のものです。

課題 Y 「動画を生成するプログラム」で「整った構造を持つ」プログラムを開発しなさい。

「整った構造」の定義は各自にお任せします (自分たちのレベルに合った内容でよい)。レポートを重視するので、どのように整っているかをしっかり書いてください。プログラムは当然グループ内で同一となりますが、レポートは各自でお願いします。レポートは次の順で記述してください。

0. 表紙 — 学籍番号＋氏名、グループメンバーの学籍番号＋氏名 (1〜2 名)、提出日付。
1. 構想・計画・設計 — どのような構想でプログラムを企画したか、プログラムはどのように設計したか。
2. プログラムコード — 必ず動作するものを提出してください。
3. プログラムの説明 — プログラムのどの部分が何をしているかの説明をお願いします。
4. 生成された動画 — アップロードで提出してください。プログラムコードと動画が一致していること。レポートにはどのような動画という説明を書いてください。
5. 開発過程の説明 — 誰が何を分担し、どのような過程を経てプログラムが完成したか。各作業の日時と担当者の記録があるとよい。
6. 考察 — 課題をやって分かったことや感想など。
7. 以下のアンケートの解答。

    - Q1. うまく分担して課題プログラムを開発できましたか。
    - Q2. 複数で分担する際に注意すべきことは何だと思いましたか。
    - Q3. リフレクション (今回の課題で分かったこと)・感想・要望をどうぞ。

生成する動画についてはクレジットつきでネットや会合等で紹介することがありますので、公序良俗に反する (ネット等に掲示できない) 動画を生成することはやめてください。