C言語班プロジェクト活動報告書

メンバー

- 1回生
  - 山本 京介
  - 佐田 淳史
  - 佐久間 智也
  - 川﨑 秀昌
  - 西村 雅貴
  - Park Jooinh
- 2回生
  - 奥川 莞多
  - 中尾 龍矢
- 3回生
  - 西見 元希
  - 坪倉 奏太

目次

はじめに

文責：西見 元希

この報告書は2020年度通年プロジェクトであるC言語班の活動報告書である．このプロジェクトの目的はC言語を学ぶことでC言語の流れを組む現在のほぼすべてのプログラミング言語の習得を容易なものとしつつ，ハードウェアに近い特性を活かしてOSの仕組みについての理解を深めるというものであった．

本プロジェクトは対象として主に新入生を想定しており，春学期は基礎文法のみを扱う予定であったが，上回生が積極的に参加し後述する外部資料をより充実なものとしたり活動内容について深堀りのできる質問をしたりするなど，かなりの活発性が見られた．感染症対策により大学での活動が制限されオンラインでの活動が主となった今年度において，上記のような積極的な活動状況は非常に喜ばしいことであったと言える．

なお，本プロジェクトの活動成果は外部WebサービスScrapbox (https://scrapbox.io/rcc-project2020-C/)により詳細にまとめられているため本報告書に掲載しきれなかった内容に関してはそちらを参照されたい．

活動の概要

文責：西見 元希

本項ではC言語班におけるプロジェクト活動の詳細を述べる．本プロジェクトでは新入生が多く参加することが予想されたため，予備知識無しで活動をスタートすることができるよう春学期の間は上回生が新入生に基礎文法を指導するという形での活動を行った．また，秋学期については春学期で学習した文法事項の知識を活用し，各自で実践的なプログラムの作成およびその発表を行った．

活動は春学期は週2回の講義形式で行い，秋学期は定例会議後のZoomをそのまま利用して各自の進捗の確認を行った．秋学期は会員のモチベーションが下がり活動中に進捗報告が行われることは頻繁ではなかったが，例年の活動を考慮すると平常通りの活動成果は得られていると考えられる．

次章からは活動内容の詳細を報告する．内容は以下の3点である．

- C言語文法
- Shell自作
- UNIXコマンド群自作

C言語文法

文責：西見 元希

本項では春学期に行ったC言語の基礎文法についての詳細を報告する．なお，学習内容すべてを報告すると冗長であるためC言語の難関とされるポインタに関する学習内容を特筆する．

アドレス演算子と間接演算子

文責：Park Jooinh

「&」は変数のアドレスを求める演算子で，「*」演算子は変数の中身が指すメモリーアドレスの値（Location Value）を返す演算子である．「&a」は変数aのアドレス「*(&a)」はaの値になる．

```
int a=10;
printf("aの値は%d",a);
    //=> aの値は10
printf("aのアドレスは%p\n",&a);
    //=>aのアドレスは0x16ba5f6fc(個人差あり)

int *b=&a; //bはポインタであり，aのアドレスが入っている
printf("bの値は&a，つまりaのアドレスの%p",b);
    //=> bの値は&a，つまりaのアドレスの0x16ba5f6fc(個人差あり)
printf("*bの値はaの値の%d\n",*b);
    //=>*bの値はaの値の10
```

「*」を付けてポインタの中身を取り出すことを「参照外し」または「参照はがし」と呼び，上のような場合の*aは変数bのエイリアス（別名・あだ名）として扱われる．

ポインタ型

文責：Park Jooinh

ポインタ型は変数のアドレスを保持するデータ型である.「*」で記述し,フォーマット指定子には%pが使われる.宣言に使われる「*」は型名の一部,printfに使われる「*」は参照外し演算子である.

```
int a=10; float b=1.0; double c=2.0; char d='a';
int *A=&a; float *B=&b; double *C=&c; char *D=&d;
printf("アドレス=%p,値=%d\n",A,*A); //アドレス=0x16f23f6fc,値=10
printf("アドレス=%p,値=%f\n",B,*B);//アドレス=0x16f23f6f8,値=1.000000
printf("アドレス=%p,値=%f\n",C,*C);//アドレス=0x16f23f6f0,値=2.000000
printf("アドレス=%p,値=%c\n",D,*D);//アドレス=0x16f23f6ef,値=a
```

ポインタは中に入っているアドレスが何を指しているのかが重要である.ポインタ型はどのタイプかに関わらず8バイト分のアドレスが入っている.
しかし,ポインタとポインタが指しているアドレスに入っているデータは同じ型である必要がある.例えば,int型ポインタにchar型のデータが入っているアドレスを入れようとすればエラーになる.

NULLとNULLポインタ

文責:Park Jooinh

何の変数も指していないポインタはNULLポインタという.NULLポインタを参照し,それが指す変数への読み書きを行おうとすると,セグメンテーション違反が発生し,プログラムは正常に動作しない.

```
if(i) //何かを指している必要があるポインタ「i」
    puts("プログラムが正常に動作するとき");
else //何も指してないとif(NULL)になるのでelseに分岐
    puts("エラーがあります");
```

`int *i=NULL;` のとき,if(i)=if(NULL)=if(0) になりif文が実行されない.ただしif(i)の場合は参照先が変わればif文が実行されるので必ず何かを指している必要があるポインタのエラー処理を作ることができる.

構造体へのポインタとアロー演算子

文責:Park Jooinh

構造体が大きくなれば関数に値を渡すとき時間が掛かりスタックも使う.よってこれを避けるために構造体のポインタを使う.このポインタの参照する構造体のメンバへアクセスするときはアロー演算子「->」を使う.

int型のidと構造体型のscoreのメンバを持つ構造体student,それを指すポインタspを想定する.

```
Student student = {1, {90, 95, 87}};
    Student *sp=&student;
    printf("idは%d=%d, スコアは%d=%d\n", student.id, sp->id,
            student.grade.math, sp->grade.math);//idは1=1, スコアは95=95
```

また，(*sp).score.math のようにstudentのエイリアスとして使うこともできる．

mallocとfree

文責：西村 雅貴

関数内で変数や配列を宣言するとスタックにメモリ領域が確保されるため，その領域の大きさを実行時に決定することはできないが，malloc関数を使うことで自由なサイズの領域を動的に確保することができる．動的に確保したメモリ領域は自動では解放されないのでfree関数を使い不要となった領域を解放する必要がある．

メモリ領域を動的に確保するメリットとして以下のコードのように関数外の変数のアドレスを参照できる点がある．

```
#include <stdio.h>

int *hoge_func() {
        int *x = (int *)malloc(sizeof(int));
        *x = 10;
        return x;
}

int main(void) {
        int *hogehoge = hoge_func();

        free(hogehoge);
        return 0;
}
```

関数外の変数のアドレスを扱う場合，単に変数を宣言しただけではその関数終了時にメモリが解放されてしまうのでmalloc関数を使いメモリ確保を行うべきである．

ダブルポインタと二次元配列

文責：西村 雅貴

ダブルポインタとはポインタのアドレスを格納するポインタであり，ダブルポインタの参照しているポインタがさらに参照している変数へアクセスすることができる．

一次元配列はポインタによって代替することができたが，ダブルポインタはポインタの連なりであるため二次元配列を代入したりすることはできない．

```
int a[3][4] = {{0, 1, 2, 3}, {10, 11, 12, 13}, {20, 21, 22, 23}};
int **dp = (int **)malloc(3 * sizeof(int *));

// dp = a; => セグメンテーション違反

for (int i = 0; i < 3; i++) {
    dp[i] = a[i];
}
```

このようにmallocでメモリを確保し，先頭アドレスのみのポインタに配列の値を受け取れるようにしてから代入する必要がある．

ポインタを引数にとる関数

文責：西村 雅貴

関数内に引数として値を渡す時，渡された元の変数のアドレスと関数内での引数のアドレスは異なるため，引数の値が変更されたとしても元の変数は変わらない．そこで，ポインタを引数にとることで参照するアドレスが同じになるため元の変数に変更を加えることができる．C言語において，このように変数のアドレスを渡すことを参照渡しという．

また，引数に配列の先頭アドレスを渡すことで違う関数で同じ配列を共有することができる．構造体を扱うときは関数の引数を減らすことができるというメリットもある．

4つのメンバを持つ構造体を引数にするとき，a,b,c,dをそれぞれ渡すよりfuncで引数を構造体へのポインタに指定して渡す方が良い．

```
  struct st  x;

  x.a = 10;
  x.b = "aaa";
  x.c = 12.12;
  x.d = 21.21;

  func( &x );                    //構造体を使った引数
  func2( x.a, x.b, x.c, x.d );  //構造体を使わない引数
```

線形リスト

線形リストはデータの列を扱うためのデータ構造である．しかし，一次元配列のようにメモリ上に順番に並べるのではなくポインタで繋げたものである．線形リストは主に識別キー（任意），一つの項目が持つべきデータ，次の項目へのポインタで構成される．

```
typedef struct __sample{
    int id; //動的データなので識別キーで管理すると便利
    char data1; //この項目が持つべきデータ1~N
    ...
    int dataN;
    struct __sample *next; //同じ構造体型のポインタ
} Sample;

```

上記のようなデータとポインタのペアを総称してノードと呼ぶ. nextは次のノードの先頭アドレスを指している. 次の項目がない場合はNULLを指すNULLポインタになる.
線形リストでは各ノードのためのメモリはヒープに確保される. よってメモリ容量の許す限りノードを増やすことができる. また, ポインタを書き換えるだけでノードの挿入・削除ができるという利点を持つ. しかし, ノードにアクセスするときに先頭のノードから順にたどる必要があるという欠点もある.

スタック（Stack）・キュー（Queue）

スタックとキューはどちらもノードを一列に並べるデータ構造である. スタックはLIFO（Last In First Out）, つまり後入れ先出しの簡単なデータ構造である. スタックにデータを入れることをPUSH, 取り出すことをPOPと呼ぶ. PUSHしたデータはスタックの先頭（Top）になり, POPすると先頭のデータが取り出される.

キューはFIFO（First In First Out）, つまり先入れ先出しのデータ構造である. 両端キュー, 優先度付きキュー等の種類も存在する. キューにデータを入れることをエンキュー（Enqueue）, 取り出すことをデキュー（Dequeue）と呼ぶ. エンキューしたデータは末尾（BackまたはRear）から入り, 先頭（Front）からデキューされる.

スタックは構文解析器を実装したりデータを探索したりする際によく使われる. 逆にキューは順次処理を扱うときに使われる.

Shell自作

文責：西見 元希

本プロジェクト活動の一貫としてShell自作を行った. 後述するUNIXコマンド自作とともにC言語における実践的な開発を経験し, プロセスの管理を通してOSへの理解を深めることを目的としている.

動作原理

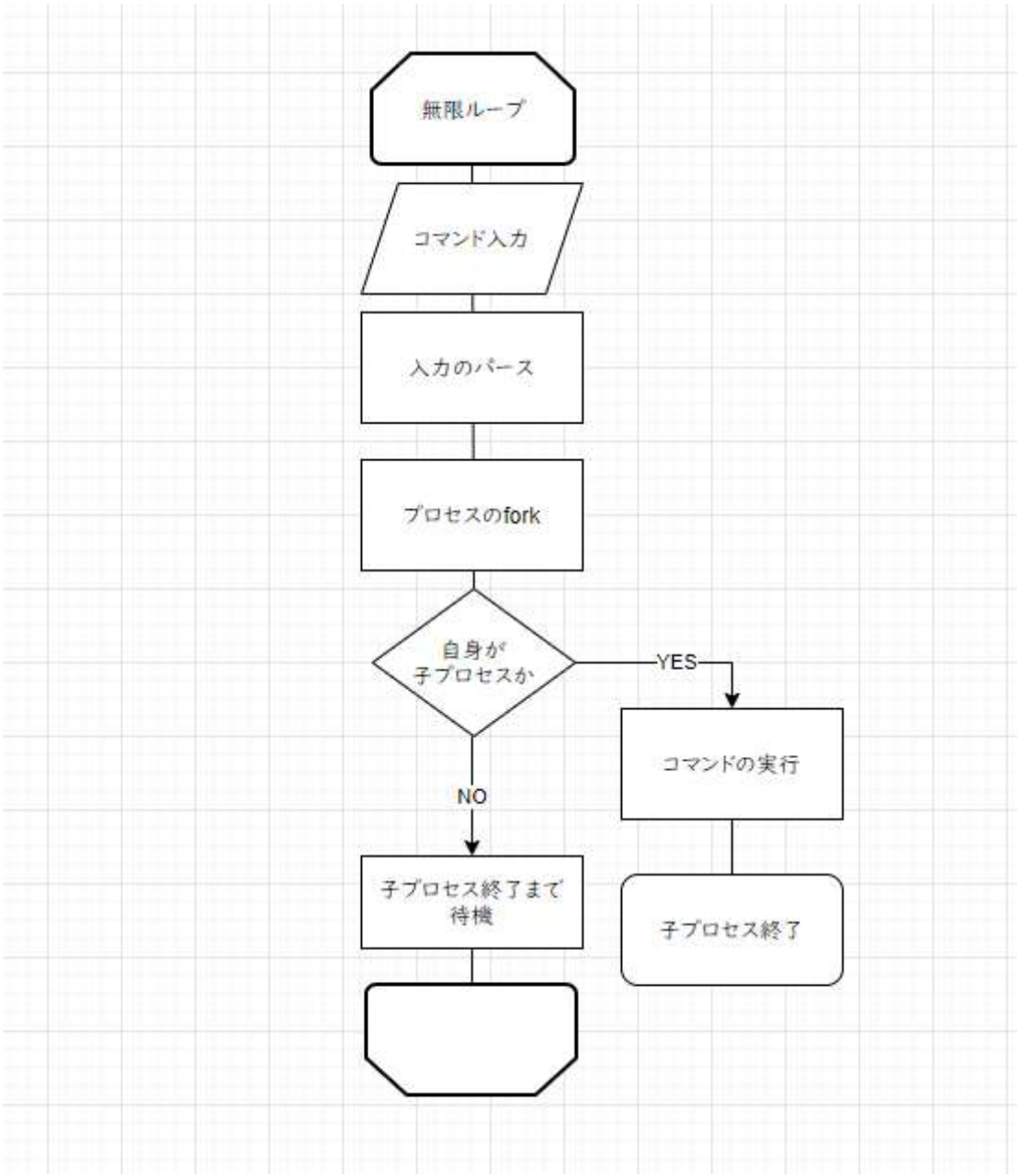
文責：西見 元希

作成したShell(以下rcshと記述)は以下の大別された段階をループすることで動作する.

1. コマンド入力とパース

2. コマンドの実行
3. コマンド終了の待機

以下の図に詳細をフローチャートで示す.



まず, rcshは標準入力からコマンド名や引数を受け取り, それをパースする. コマンドの実行には文字列の配列を用意する必要があり, 標準入力から受け取った文字列を空白文字で区切る必要があるためである. 入力のパースは少々複雑な工程を経るがrcshの動作に関しては本質的ではないためここでは省略する.

パースが完了し実行するコマンドと引数から成る文字列の配列が完成したらrcshはプロセスのforkを行う. forkはプロセスが自身のコピーを作成するためのシステムコールであり, これをプロセスがOSに対して発行するとOSはそのコピーである子プロセスを生成する. 子プロセスには現在の親プロセスのメモリの状態などもコピーされるため, 子プロセスも親プロセスもforkの後全く同じ処理を行うことになる. したがってforkした後に自身が子プロセスであるかの条件分岐を行い, 自身が子プロセスであっ

たときのみコマンドの実行処理を行うという記述が定石である.コマンドの実行にはexecというシステムコールを用い現在のメモリ状況を破棄して新たな処理(ここでは先ほどパースしたコマンド)を開始する.

自身が親プロセスであった場合は子プロセスの終了を待機する.待機にはwaitというシステムコールを用いており,子プロセスが終了したというシグナルを受理するまで処理を停止させ,シグナルを受け取ったら子プロセスの利用していた計算資源を解放するという処理を行う.

プロセスのforkと条件分岐部分は以下のソースコードから拡張することで実装した.

```
// fork関数は子プロセスのプロセスIDを返すが自身が子プロセスの場合0を返す .
pid_t pid = fork();
if(pid < 0) {
    perror("rcsh: ");
    exit(1);
}
if (pid == 0) {
     puts("I am child proc");
     execv(argv[0], argv);

     // execで完全にプロセスが置き換わるため本来ここには到達しない
     perror("rcsh: ");
     exit(1);
}

int status;
wait(&status);
```

実装における工夫

文責:西見 元希

rcshと後述の自作UNIXコマンド群は本会GitHub内で同一リポジトリを利用していたため,rcshから自作したコマンド群を優先して呼び出すという実装を行った.この際,コマンド自作において特に指定をしていなかったため各コマンドプログラムはmain関数の実行によって記述されており,rcshからのファイルのインクルードによって結合することができなかった.したがってビルドシステムであるmakeとrcsh内でのコマンド検索により,指定したディレクトリ内のバイナリと同じ名前のコマンドが入力されていた際にそちらを実行するという実装を行うことで解決した.結果的にはMakefileの記述方法やC言語のAPIの知見となったため,これを工夫点として挙げる.

以下はプロジェクトのルートに配置されているMakefileである.

```
subdirs := $(shell find . -maxdepth 2 -path "./commands/*")

.PHONY: all $(subdirs) clean
all: $(subdirs) rcsh

$(subdirs):
	$(MAKE) -C $@ $(MAKECMDGOALS)

rcsh: rcsh.c
	gcc rcsh.c -Wall -Wextra -g -o rcsh

clean: $(subdirs)
	rm rcsh

```

今後の展望

文責：西見 元希

今後の展望として，以下の3点が挙げられる．

- バックグラウンドプロセスの管理機能
- パイプの実装
- スクリプト言語化

バックグラウンドプロセスの管理機能については，現状バックグラウンドで実行しているプロセスの一覧などを出力する(jobsコマンド)ことができないため，その実装が求められる．また，バックグラウンド実行中のプロセスをフォアグラウンドで待機するfgコマンドなども併せて実装していく．

残り2点に関しては実現するには難易度が高いと予想されるが，今後の拡張性の方向性を示すものとして記述した．パイプの実装については標準入出力を介したプロセス間通信を簡単にする機能としてrcshに実装したいと考えている．スクリプト言語化については，さらに難易度が高まるが，その分実際によく利用されているシェルプログラムであるbashやzshへの理解が深まると思われる．

参考文献

write-a-shell-in-c.
https://brennan.io/2015/01/16/write-a-shell-in-c/ (https://brennan.io/2015/01/16/write-a-shell-in-c/)

瀧本栄二, 西村俊和.「ネットワーク開発実験2019年度後期レジュメ」. 立命館大学, 2019

UNIXコマンド群自作

文責：西見 元希

本プロジェクト秋学期のメイン活動としてUNIXコマンドの自作を行った. 内容としては班員それぞれが好きなコマンドを選び, C言語で簡単な実装を行うというものであった. 春学期に学んだ基礎文法の実践の場を作るとともに, Gitを用いた集団開発の経験を得るという意図で実施した.

本報告書で記述するコマンドは以下の通りである.

- grep
- wc
- diff
- ping
- echo
- stat
- mv

grep

文責:川﨑 秀昌

仕様

検索キーワードと検索対象の文字列を引数で渡すとマッチした行を出力する.

実装

オプション(v,i,n)を実装した.

オプションの指定

始めにオプションを指定する際に - を使用しているかを検知することによってオプションを設定するかを検出している. その後, どのオプションが入力されたかを確認し, それをswitch文でインクリメントし, フラグを立てることで複数のオプション指定も可能にした.

```
int opt_i = 0;
int opt_v = 0;
int opt_n = 0;
for(i = 0;i<argc-2;i++){
  if(argv[i][0] == '-'){//文字'-'の検出
    int j=1;
    while(j<sizeof(argv[i])){
      switch(argv[i][j]){]
      //Optipon i
        case 'i':
          opt_i++;
          break;
      //Option v
        case 'v':
          opt_v++;
          break;
      //Option n
        case 'n':
          opt_n++;
          break;
      }
      j++;
    }
  }
}
```

vオプション

vオプションを付けると検索キーワードがマッチしなかった行を出力する.

文字列の検出,取得

ファイルから取得した文字列に対しstrstr関数を使用し,同じ文字列があった行は `MatchLine` 構造体に代入,その他の行は `UnMatchLine` 構造体に代入することによって文字列の内部保存や区別を行い,vオプションの処理を可能にした.

```
//引数からの文字列の取得
for(i=0; i<sizeof(argv[argc-2]);i++){
  GetLine[i] = argv[argc-2][i];
}
//devided MatchLine and UnMatchLine from LineFromFile//
while(fgets(LineFromFile, N, file) != NULL){
//UnMatchLine process//
  if(strstr(LineFromFile,GetLine) == NULL){
    unmatch_line[count].line_number = line_count;
    strncpy(unmatch_line[count].line,LineFromFile,N);
    count++;//UnMatchLine構造体の配列
//MatchLine process//
  }else if(strstr(LineFromFile,GetLine) != NULL){
    match_line[se_count].line_number = line_count;
    strncpy(match_line[se_count].line,LineFromFile,N);
    se_count++;//MatchLine構造体の配列
  }
  line_count++;//実際のファイルに入っていた行数カウント
}
```

iオプション

iオプションを付けると文字の大文字，小文字を無視して検出する．そのため，検索キーワードである `GetLine` とファイルの各行を返す `LineFromFile` をすべて小文字に変換することで大文字，小文字の区別をなくし，上記の1回の文字列で検索できるようにした．

```
if(opt_i){
  for(i=0;i<N;i++){
    GetLine[i] = tolower(GetLine[i]);
    LineFromFile[i] = tolower(LineFromFile[i]);
  }
}
```

文字列の出力

構造体に入っている行番号と文字列を変数 `line` に代入し，出力することによってその前のvオプションの有無でそれぞれ出力していた冗長性を排除した．

```
 while(i < max(count,se_count)){
  if(opt_v){
    number = unmatch_line[i].line_number;
    strncpy(line,unmatch_line[i].line,N);
  }else{
    number = match_line[i].line_number;
    strncpy(line,match_line[i].line,N);
  }

  printf("%s",line);

  i++;
 }
```

nオプション

　行番号を出力するオプションである.この実装では各行とその行番号を構造体で保存しており,文字列を格納していない構造体の行番号は0で初期化されているため,全ての行を出力した後さらに構造体配列を読めば行番号が0であるためwhile文からのbreakをしている.

```
if(opt_n){
  if(number == 0){
    break;
  }
  printf("%6d  ", number);
}
```

反省と感想

　オプションCの実装に手間取ってしまった上にできないという結果になってしまった.他にも合致した文字列の色を変えるcolorオプションの追加ができなかったのに加えて,構造体を使用してコードの綺麗さを追求したが最終的には行数が大きくなってしまった.これから読みやすいコードを書けるように精進したいと考えている.しかし,実際によく使用するコマンドを自作することによって内部処理など多くのことを学べることができたため,自分のシェル自作の場合でも多くのコマンドを自作していこうと思う.

WC

文責:中尾 龍矢

機能

　wcコマンドはファイルを指定することで,そのファイルの情報を出力する.

実装

実装には大まかに「オプションの解析」と「各種情報を計測」の2ステップを踏むことで実現した.

オプションフラグの管理

まず初めに,オプションが指定されたか否かはフラグで管理することにする.そこで各フラグをまとめた構造体を用意する.(wcコマンドのオプションはc, m, L, wの4種.ファイルの文字コードはutf-8を想定している)

```
//オプションのフラグをまとめた構造体
typedef struct {
        unsigned int c:1;       //オプションに-cが選択されているか
        unsigned int m:1;       //オプションに-mが選択されているか
        unsigned int L:1;       //オプションに-Lが選択されているか
        unsigned int w:1;       //オプションに-wが選択されているか
        unsigned int no_option:1;       //オプションが何も選択されていない
} options;
```

オプションの解析

一つ目の「オプションの解析」について,考えるべきは「オプションの指定あり」「オプションの指定なし」の二つである.

オプション指定なし

```
 //オプションの解析
if(argc == 2 && argv[1][0] != '-'){
//オプション指定がなく，ファイルのみの指定のとき
    opts.no_option = 1; //no_optionのフラグを立てる
    target_file = argv[1];  //ファイル名を保持
}
```

「オプション指定なし」の場合は引数の個数argc=2かつ引数の文字列argv[1][0]!="-"(オプションは指定されておらず,ファイル名のみ指定されている)という条件で検知している.

オプション指定あり

```
//オプションの指定があるとき
//getoptを用いてどのオプションが指定されたかを解析
while((ch = getopt(argc, argv, "cmLw")) != -1){
    switch(ch){
        case 'c': //cオプションのフラグを立てる
            opts.c = 1;
            break;
        case 'm': //mオプションのフラグを立てる
            opts.m = 1;
            break;
        case 'L': //Lオプションのフラグを立てる
            opts.L = 1;
            break;
        case 'w': //wオプションのフラグを立てる
            opts.w = 1;
            break;
        default: //間違ったオプションを受け取ったとき
            printf("Try 'wc --help' for more information.\n");
            exit(1);
    }
}
target_file = argv[optind]; //ファイル名を保持
```

一方,「オプション指定あり」の場合はunistdライブラリのgetopt命令を用いてオプションの解析を行っている. この命令は第一引数argcに第二引数argv, 第三引数にオプションに指定可能な文字を文字列として指定して使う. これでoption characterがchに格納され, それに付くoption argumentの先頭アドレスがoptargに格納される. optindは次に解析するargvのindexであるため, 必ず最後にoption argumentを書くことを約束すれば, 全てのoption characterを解析した後, argv[optind]にoption argumentがある.

ここでコマンド引数の各部分の読み方について, wc -c file.txtと書いたとき, cがoption character, file.txtがoption argumentとしている.

各種情報を計測

次は, 各オプションに対する出力に必要な各情報を計算する.
各オプションでの必要な情報は, -cではファイルのバイト数, -mはファイルの最大バイト数の行のバイト数, -Lはファイルの単語数, -wはファイルの行数である. ただし, オプション無しの場合はファイルのバイト数, ファイルの行数, ファイルの単語数の三つが必要である.
これらを計算する各関数の実装を以下に示す. ただし, ファイルを開くまでの処理はすべて同じであるため, ファイルを開いた後の処理のみを掲載する. 全ての関数において, ファイルポインタはfp, 返り値はcntとしている.

ファイルのバイト数をカウント

```
while(fgetc(fp) != EOF) cnt++;  //1byte毎にカウントアップ
```

1byte毎に最後まで読み込みながらカウントすればよい.

ファイルの文字数をカウント

```
while((c = fgetc(fp)) != EOF) {
    //utf-8のマルチバイト文字の2バイト目以降の文字を検出したらカウントをスキップ
    if((c & 0xc0) == 0x80)continue;
    cnt++;
}
```

utf-8の文字のバイト数は固定ではなく，1～6byteで1文字を表現している(※実際は1～4byteまでしかないので5，6byteの文字は考えなくてよい)．その際に，多バイト文字の2byte以降は，上位2bitが10と決められている．なので1byte毎に読み込み，上位2bitが10以外のものがあればカウントすればよい．

ファイルのバイト数最大の行のバイト数をカウント

```
while(fgets(str, N, fp) != NULL){ //行毎に読みこむ
    row_cnt = count_character_str(str);
    if(cnt < row_cnt)cnt = row_cnt;  //最大値を更新
}
.
.
.
//文字列の文字数を計算をカウント
int count_character_str(char *str){
        int row_cnt = 0;
        char *ptr = str;
        while(*ptr != '\0' && *ptr != '\n' && *ptr != EOF){
        //その行のバイト数をカウント
                row_cnt++;
                ptr+=1;
        }
        return row_cnt;
}
```

行ごとに読み込んだ文字列を，count_charcter_str関数に渡し，先頭から改行コードもしくはEOFが来るまでのバイト数をカウントを返してもらう，それが以前の最大バイト数よりも多ければcntを更新している．

ファイルの単語数をカウント

```
while(fgets(str, N, fp) != NULL){ //行ごとに読み込む
    char *word = strtok(str, TOK_DELIM); //区切り文字毎にカウントアップする
    while(word != NULL) {
        cnt++;
        word = strtok(NULL, TOK_DELIM);
        }
}
```

行ごとに読み込み，区切り文字で何度区切れるかをカウントすればよい．

ファイルの行数をカウント

```
while(1){
    c = fgetc(fp); //1byte毎に読み込む
    if(c == '\n')cnt++;  //改行コードがあればカウントアップ
    if(c == EOF)break; //EOFを検出したら終了
}
```

1byteごとに読み込み，改行コードの出現回数をカウントすればよい．

反省

今回のコマンド自作ではコマンドライン引数などの実際によくお世話になっている機能を自分で作ることが新鮮に感じた．

そして，できる限り実際のlinuxでのwcコマンドと同じ結果になるように実装した．しかし，このプログラムでは，実際のwcコマンドのように複数のファイルを同時に指定する事ができない，そのほか文字コードがutf-8のみにしか対応していないなどの反省点がいくつかある．まだ改善の余地は見られるので精進したいと思う．

余談では有るがwindows(gcc)での動作も確認されたので，windowsでのオリジナルコマンド実装にも挑戦したいと思う．

diff

文責：山本 京介

diffとは

diffは2つのテキストファイルを比較し，異なる箇所(差分)と該当する行番号を表示するコマンドである．書式は `diff [オプション] ファイル1 ファイル2` で，オプションを指定すると比較対象のファイルを2列に分けて表示できたり，特定の違い(大文字小文字など)を無視したりできる．

自作diffの仕様

自作diffではまず差分を表示できるようにした．差分検出はファイル1とファイル2を1行ずつ，ファイル1の最終行まで比較し，一致しない行があればそれを表示するという流れで行う．一致しない行の検出には，別の文字列に置き換えられた場合のみならず，新しい行が挿入された場合や行全体が削除される場合を考慮する必要がある．特に行の増減に関しては，1行ずつ比較する方法では検知できず問題が生じる．この問題を回避するため，その変更が行数の増減を伴うものであったかどうかを判定し，増減があれば比較対象とする行を前後するようにした．

行全体が削除された場合であるかどうかの判定

```
/*searchA，Bはそれぞれ現在比較しているファイル1，2の行番号
num1，2はそれぞれファイル1，2の行数*/
for (int i = searchA + 1; i < num1; i++) {
    if (!strcmp(str1[i], str2[searchB])) {
        for (int j = searchA; j < i; j++) {
            printf("<%s", str1[j]);
        }
        searchA = i; //削除された行を飛ばす
        delete = 1; //行全体が削除されたことを示すフラグ
        break;
    }
}
```

ファイル2のsearchB行目と一致するものがファイル2のi行目(searchA<i<num1)にあるならば，searchA行目からi行目において行全体の削除が行われたと判定し，削除された行を差分として表示する．この場合，ファイル1では削除された行を飛ばして比較を続行し，ファイル2ではそのまま続行する．

新しい行が挿入された場合であるかどうかの判定

```
for (int i = searchB + 1; i < num2; i++) {
    if (!strcmp(str1[searchA], str2[i])) {
        for (int j = searchB; j < i; j++) {
            printf(">%s", str2[j]);
        }
        searchB = i; //挿入された行を飛ばす
        insertion = 1; //挿入されたことを示すフラグ
        break;
    }
}
```

ファイル1のsearchA行目と一致するものがファイル2のi行目(searchB<i<num2)にあるならば，searchB行目からi行目において行の挿入が行われたと判定し，挿入された行を差分として表示する．この場合，ファイル2では挿入された行を飛ばして比較を続行し，ファイル1ではそのまま続行する．

削除とも挿入とも判定されなかった場合

この場合，文字列の書き換えだと見なすことができ，行数の変更を伴わないので，単に変更箇所を差分として表示するだけでよい．

```
if(!delete && !insertion) printf(">%s", str2[searchB]);
```

ファイル2に最後まで比較されない行がある場合

この場合，ファイル1の最終行より後の行に文字列が追加されたと見なすことができ，それらを全て差分として表示する．

```
for(int i = searchB; i < num2; i++){
    printf(">%s",str2[i]);
}
```

ping

文責:佐久間 智也

基本機能

ICMPを対象hostとの間でやりとりすることでノードの到達性を調べるコマンドである. pingコマンドの基本的な流れは, Raw socketを生成し, ICMPエコーリクエストを送信, ICMPエコーリプライを受け取るというものである.

自作pingの仕様

今回の実装では返信元アドレスの検証と多くのオプションを見送った. 実際のpingコマンドは使用者が指定しない限りICMPパケットを送信し続けるが, このpingコマンドはオプションで回数を指定しない限り1回のみ送信する.

Raw socketの生成

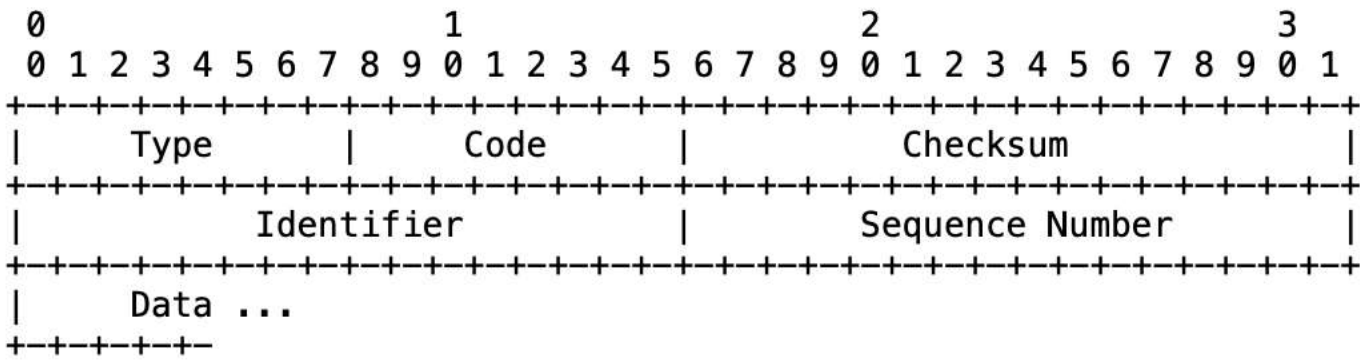
Raw socketの生成には, socketシステムコールを使用する.

```
/*socket()の第1引数のAF_INETはネットワークアドレスの種類を指定し，
第2引数のSOCK_RAWはraw network protocolを提供する．
第3引数はどのプロトコルでsocketを開くかを指定する．*/
int make_raw_socket() {
    int s = socket(AF_INET, SOCK_RAW, IPPROTO_ICMP);

    if (s < 0) {
        perror("failed make socket");
        exit(EXIT_FAILURE);
    }

    return s;
}
```

ICMPヘッダの生成

ICMPヘッダはRFC792では下記のように定義されている.
typeにはエコーリクエストの場合は8, リプライの場合は0が入る.
IdentifierにはプロセスIDを入れる. Sequence Numberは同じIdentifierの場合に

それぞれを識別するための番号であり，送信ごとにインクリメントする．




```
/*struct icmphdrは#include <netinet/ip_icmp.h>に定義されている*/
void set_icmphdr(uint8_t type, uint8_t code, uint16_t id, uint16_t seq,
  struct icmphdr* icmphdr) {

    memset(icmphdr, 0, sizeof(struct icmphdr));

    icmphdr->type = type;
    icmphdr->code = code;
    icmphdr->checksum = 0;
    icmphdr->un.echo.id = id;
    icmphdr->un.echo.sequence = seq;

    icmphdr->checksum =
      checksum((unsigned short*)icmphdr, sizeof(struct icmphdr));
}
```

チェックサムの計算

チェックサムの計算では，16bitごとの1の補数和に対してさらに1の補数を取り，最後に反転を行う．計算時はチェックサムフィールドに0を入れておく．

```
uint16_t checksum(unsigned short* buf, int size) {

    unsigned long sum = 0;

    while (size > 1) {
        sum += *buf;
        buf++;
        size -= 2;
    }

    if (size == 1)
        sum += *(unsigned char*)buf;

    sum = (sum & 0xffff) + (sum >> 16);
    sum = (sum & 0xffff) + (sum >> 16);

    return ~sum;
}
```

ICMPヘッダの取得方法

　対象hostから受信したパケットにはIPヘッダも含まれるため，recv_iphdr->ihlによってIPヘッダ長を取得し，シフト演算を行うことでIPMCヘッダの先頭ポインタを得る．

```
struct iphdr* recv_iphdr;
    recv_iphdr = (struct iphdr*)buf;

    struct icmphdr* recv_icmphdr;
    recv_icmphdr = (struct icmphdr*)(buf + (recv_iphdr->ihl << 2));

```

反省

　今回は，送信回数を指定するオプションのみしか実装できず，名前解決機能を実装することができなかった．加えて，返信元アドレスの取得に失敗するバグがあり，正確な検証に支障がでた．今後，バグの解消と名前解決，更なるオプションの実装をしていきたい．

参考文献

ICMPヘッダのチェックサムを求めるメモ
https://qiita.com/kure/items/fa7e665c2259375d9a81
(https://qiita.com/kure/items/fa7e665c2259375d9a81)

RFC 792 - Internet Control Message Protocol - IETF Tools
https://tools.ietf.org/html/rfc792 (https://tools.ietf.org/html/rfc792)

Raw socket - Wikipedia - ウィキペディア
https://ja.wikipedia.org/wiki/Raw_socket (https://ja.wikipedia.org/wiki/Raw_socket)

ping - Wikipedia - ウィキペディア
https://ja.wikipedia.org/wiki/Ping (https://ja.wikipedia.org/wiki/Ping)

Internet Control Message Protocol - Wikipedia - ウィキペディア
https://ja.wikipedia.org/wiki/Internet_Control_Message_Protocol
(https://ja.wikipedia.org/wiki/Internet_Control_Message_Protocol)

Ping を作ろう | エンジニアブログ | GREE Engineering
https://labs.gree.jp/blog/2010/06/336/ (https://labs.gree.jp/blog/2010/06/336/)

簡単なpingの作成（ICMPの送受信）:Geekなページ
https://www.geekpage.jp/programming/linux-network/simple-ping.php
(https://www.geekpage.jp/programming/linux-network/simple-ping.php)

echo

文責:佐田　淳史

概要

echoコマンドは引数を受け取りそのデータを出力するという単純なコマンドである.以下,オプションを排除したシンプルな実装を示す.

```
#include <stdio.h>

int main(int argc, char **argv){
    for(char **p = argv + 1; *p; p++)
    printf("%s\t", *p);
    printf("\n");
}
```

上記のプログラムを実行すると以下のような結果を得る.

```
>>> ./echo hogehoge
hogehoge
```

引数で与えた文字列(hogehoge)が呼応する形で返ってきているのでechoコマンドとして成功していることが分かる.

オプションの実装

本echoコマンドは-eオプションを持っているためGNU派生のechoコマンドになっており,printfが採るようなescape sequenceを解釈して表示する.すなわち,`\n` や `\t` はオプションなしの場合ではそのまま表示されるが,-eを用いて実行することで改行文字やタブ文字として表示される.

以下にソースコードを載せ簡単に説明を補足する.
引数はargvに渡されるのでargvを順に出力する.argvの0番目の要素はコマンド自身になっているためそれを除外することによって1番目の引数以降を表示する.

```c
#include <stdio.h>
#include <string.h>

static void unescape(char *p){
    for (; *p; p++){
        if(p != '\\' || p[1] == '\0' ){
            printf("%c", *p);
            continue;
        }
        p++;
        if (*p == 'n')
            printf("\n");
        else if (*p == 't')
            printf("\t");
        else printf("\\%c", *p);
    }
}
int main(int argc, char **argv) {
    if(argc == 1)
        return 0;

    char **args = argv + 1;
    if (strcmp(args[0], "-e") == 0){
        args++;
        for (char **p = args; *p; p++){
            if (p != args)
                printf(" ");
            unescape(*p);
        }
    } else {
    for(char **p = args; *p; p++)
        printf("%s%s", (p == args ? " " : ""), *p);
}
printf("\n");
return 0;
}
```

想定される実行結果と"**-e**"オプションをつけたことによる違い

```
a
f
rtyhgf ii
```

となり改行\nやタブ文字\tの表示に成功したことになる.

stat

文責:奥川 莞多

statコマンドは指定したファイルまたはディレクトリの状態を標準出力に表示するコマンドである. 自分は本活動でこのstatコマンドを作成した. statコマンドの書式は `./stat [-L] [-c format] [-t] file` となっている. オプションLについては実装することができなかったが, その他機能はなるべく詳細に実装した.

自作statコマンドの外部仕様

まず, 自作statコマンドの外部仕様について記載する.

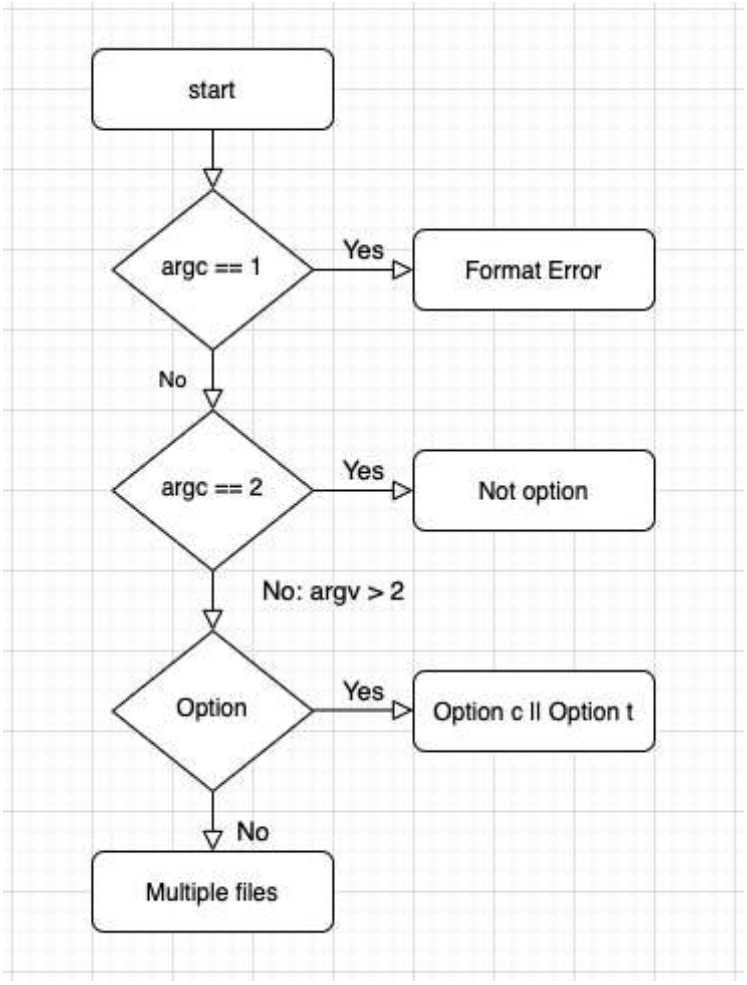
statコマンドの基本的なオプションなしの機能を実装した. 実行方法は `./stat file` である. また, 複数ファイルを指定する場合は, `./stat file file ...` となる. オプションなしの場合はラベル付きでファイルの状態が出力される.

オプションについてはtとcを実装した. オプションtは空白区切りでファイルの状態を出力する. オプションtの実行方法は `./stat -t file` である. オプションcはオプションの後にフォーマットを指定し, それに応じてファイルの状態を出力する. また, フォーマットを複数指定することができる. 本来のオプションcであれば大文字フォーマットもあるが, これは実装することができなかった. フォーマットの複数指定は実装することができた. オプションcの実行方法は `./stat -c %a file` または `./stat -c "%a %b" file` である.

自作statコマンドの内部仕様

次に, 自作statコマンドの内部仕様について簡潔に記載する.

大枠のフローチャートを以下の図に示す.



上記の図についてそれぞれ順に説明する.

コマンドライン引数が一つの場合はエラーのみのため省略する.コマンドライン引数が二つの場合について説明する.第3引数がファイル名ならば,正しい入力と認識し,出力を実行する.第3引数がオプションとなっている場合後ろにファイル名が必要なので,エラーをそれぞれの場合に合わせて出力する.前述のソースコードを以下に示す.create_pathname関数はコマンドライン引数から絶対パスを生成している.print_all関数は出力の動作をまとめている.need_argument_err関数,illegal_option_err関数はエラーを出力する.

```
if (argc == 2) {
    if (strncmp("-", argv[argc - 1], 1) == 0) {
      char opt;
      strncpy(&opt, ++argv[argc - 1], 1);
      switch (opt) {
      case 'c':
      case 'L':
      case 't':
        need_argument_err(opt);
        break;
      default:
        illegal_option_err(opt);
        break;
      }
      exit(EXIT_FAILURE);
    } else {
      char pathname[256];
      create_pathname(argv[argc - 1], pathname);
      print_all(pathname);
      return 0;
    }
  }
```

コマンドライン引数が三つ以上の場合はファイルが複数指定されている状態とオプションが指定されている状態がある.

まず,ファイルが複数指定されている場合について説明する.ファイルが複数の場合はコマンドライン引数が二つの時の動作を繰り返すだけである.このソースコードを以下に示す.

```
if (strncmp("-", argv[1], 1) != 0) {
      char pathname[argc - 1][256];
      for (int i = 0; i < argc - 1; i++) {
        create_pathname(argv[i + 1], pathname[i]);
        print_all(pathname[i]);
      }
      exit(EXIT_SUCCESS);
} else{
    ...
}
```

次に，オプションが指定されている場合について説明する．オプションはcオプションとtオプションを実装した．オプションLは実装できなかった．オプションがcの場合フォーマットが必要なため引数の数の確認を行なっている．オプションの実装はそれぞれの関数に分割した．オプションがある場合のここまでのソースコードを以下に示す．

```
if(strncmp("-", argv[1], 1) != 0){
...
} else {
    char pathname[256];
    char opt;
    strncpy(&opt, ++argv[1], 1);
    switch (opt) {
    case 'L':
      printf("tyottomuridesita");
      break;
    case 't':
      create_pathname(argv[argc - 1], pathname);
      option_t(pathname);
      break;
    case 'c':
      if (argc < 4) {
        need_argument_err(opt);
      } else {
        option_c(argv);
      }
      break;
    default:
      illegal_option_err(opt);
      break;
    }
    exit(EXIT_SUCCESS);
}
```

オプションtについてはprint_allとほとんど変わらないので省略する．

オプションcについて説明する．オプションcは要求されたフォーマットを抽出する必要がある．また，複数フォーマットを指定された場合のためにfor文で実装している．以下にソースコードを示す．

```
void option_c(char *argv[]) {
  if (strncmp("%", argv[2], 1) != 0) {
    need_argument_err('c');
    return;
  }
  char pathname[256];
  create_pathname(argv[3], pathname);
  struct stat file_info;
  if (stat(pathname, &file_info) == -1) {
    fprintf(stderr, "stat: %s: ", pathname);
    perror("stat");
    exit(EXIT_FAILURE);
  }
  char *format[13];
  int format_num = space_judge(format, argv[2]);
  for (int i = 0; i < format_num; i++) {
    char fmt;
    strncpy(&fmt, ++format[i], 1);
    switch (fmt) {
    ...
    }
  }
  printf("\n");
}
```

課題と展望

statコマンドの実装を行った上で得られた課題点と今後の展望について記載する.

自作statコマンドではオプションによって出力する書式が変わる.そのため,出力をするためのソースコードが多くなってしまっている.このソースコードを短くするもしくはファイルを分けるなどの工夫をしてリファクタリングしたいと考える.絶対パスでのファイル指定に対応できていないため対応したい.予期せぬエラーがまだあると感じているので,再確認してなるべくエラーを網羅したい.UNIX,Linuxにおいてはシンボリックリンクファイルで実行することができる.また,オプションLでシンボリックリンクを追うことができる.その動作を実装できたら非常に良いものができる.

感想

statコマンドを選んだ理由は講義でc言語のstat関数を習い,これがあれば簡単に実装できると考えたからである.しかし,甘くはなかった.まず,オプションやstatコマンド自体を理解するのに非常に苦労した.また,予期せぬ入力全てに対応するとなると,かなりの時間が必要であると感じた.さらに,細かいところまでやるとなると,全然終わらないことに気づいた.その中でも努力して自分が気づいたところの実装を行った.

一つのコマンドを実装することでそのコマンドについてよく知ることができた.また,今回の活動でより一層C言語が嫌いになった.これらの点は非常に良い経験となった.

参考文献

Webサイト：Linux基本コマンドTips122:【stat】コマンド―ファイルの属性や日付などを表示する
https://www.atmarkit.co.jp/ait/articles/1706/29/news027.html
(https://www.atmarkit.co.jp/ait/articles/1706/29/news027.html)

mv(Tsubokura)

文責：坪倉 奏太

私は本活動において，数あるUNIXコマンド群の中でファイルやディレクトリを移動やその名前の変更を行うコマンドである「mvコマンド」を自作した．なお，今回実装したmvコマンドはあるファイルを移動させたり，その名前の変更を行う機能のみを持っている．

今回実装したmvコマンドの外部仕様

まず，今回実装したmvコマンドの外部仕様について詳しく説明する．今回実装したmvコマンドは，「mv 移動するファイル名 移動する先のディレクトリ/ファイル名」という形で実行する．まず，コマンドライン引数の数が3でないとエラーメッセージを出力して終了する．また，移動するファイル名，移動する先のディレクトリ/ファイル名が存在しない場合もそれぞれエラーメッセージを出力して終了する．移動に成功した場合，移動するファイル名と移動した先の場所を表示して終了する．

今回実装したmvコマンドの内部仕様

次に，今回実装したmvコマンドの内部仕様について簡潔に説明する．今回実装したmvコマンドは，大きく以下の流れで処理が行われる．

1. コマンドライン引数の個数を調べる
2. 各ファイル/ディレクトリの存在を調べる
3. rename関数を用い移動させる

最初に，コマンドライン引数の値を調べ，3以外の値だと終了する．
次に，移動したいファイル名を `argv[1]` を引数にした読み込みモードであるfopen関数を用いて開き，その返り値がNULLポインタであれば，つまり移動したいファイルが存在しなければ終了する．
その後，三つ目のコマンドライン引数の属性をstat関数にて調べ，もしディレクトリと判断されればディレクトリ名とファイル名を結合させ，rename関数で移動したいファイルを移動させる．もしファイルと判断されれば，そのファイル名を用い，移動させたいファイルをrename関数で移動させる．

課題と展望

次に，今回の実装を通じて得られた課題点や展望について述べる．
まず，今回の実装では，ファイルの移動先が既に存在している場合での，rename関数の処理など実行している処理系に依存している箇所が多い点が課題として挙げられる．これはコードの移植性を下げることにつながるので，今後は今実行している処理系を調べてそれにより処理を分岐させるなど処理系に依存している箇所をなるべく減らすようにしていきたいと考えている．特に，移動先のファイルが存在している場合については，ターミナルを用いてユーザにファイルの移動を実行するかどうかを訊ね

るという処理を実装することも考えている.

　また, 今回実装したmvコマンドは本来のmvコマンドと比べて機能が限定的だということも課題として挙げられる. 本来のmvコマンドは, 1回に複数のファイルの移動を行うことができるのであるが, 今回実装したmvコマンドは1回につき一つのファイルしか移動できない. さらに, 本来のmvコマンドは移動したいファイルを, 移動の前に予めバックアップのためにコピーするという機能が存在するのだが, 今回実装したmvコマンドにはそのような機能は存在しない. 今後は, 本来のmvコマンドを参考にして, 複数のファイルを移動できるようにしたりするなど新しい機能の追加を行っていきたいと考えている.

感想

　最後に, 今回の実装を通じて得られた感想について述べる.

　まず, 私はC言語でのプログラミングからかなり離れていたこともあり, 今回の実装において普段利用しているPythonでの実装との違いに大変苦労した. 特に, Pythonを用いての実装では今実行している処理系の違いやポインタについて意識する機会がほとんどないため, 自分の意図している処理をなかなかうまく動かせなかったりと苦労し, 自分がいかに言語仕様, 及び普段意識していないレイヤの低い技術や分野について理解が不足しているか痛感させられた. しかしながら, そのような普段意識していない箇所を意識して行うプログラミングは一種の思考を伴うパズル的要素があり, 達成感もひとしおであった. これから先も, 今回のmvコマンドの実装で得られた経験を活かし, 様々な形のプログラミングに挑戦していきたい.

参考文献

参考文献は多岐に渡るため, 私が特に重要だと思ったものをいくらか挙げるものとする.

C言語系/「デーモン君のソース探検」読書メモ/09, mv(1)
https://www.glamenv-septzen.net/view/553 (https://www.glamenv-septzen.net/view/553)

mvでリネームができるわけをどこまで深く話せますか - Qiita
https://qiita.com/junjis0203/items/9e8f642b04d9754f1139
(https://qiita.com/junjis0203/items/9e8f642b04d9754f1139)

ファイルの情報取得 | LinuxC
http://linuxc.info/directory/directory3.html (http://linuxc.info/directory/directory3.html)

ファイル名変更, 移動, 削除, 存在確認(C言語) - 超初心者向けプログラミング入門
https://programming.pc-note.net/c/file8.html (https://programming.pc-note.net/c/file8.html)

もう一度基礎からC言語 第28回 データ構造(7)～ポインタの配列とポインタのポインタ パラメータの数と内容を取得する
https://dev.grapecity.co.jp/support/powernews/column/clang/028/page02.htm
(https://dev.grapecity.co.jp/support/powernews/column/clang/028/page02.htm)

総評

文責：西見 元希

C言語の基礎文法に関しては全員が問題なく理解できていると考えられる. 特に, C言語で難関とされるポインタについてもScrapboxにて詳細な解説記事が生成され, 丁寧な説明で進めたため1回生の理解も深まった.

次に, Shell自作については, 担当者の実装が遅れたため全員が手元で実装し動作を確認することが難しかった. これは役割の分担や事前の準備で解決すべき問題点であった.

UNIXコマンド群自作については, 班員の多くが実装できており, GitHubを用いた開発も活動面で大きな支障は生じなかった. 自分で適度な難易度のコマンドを選び自作することは全員が同じものを実装するよりも理解が深まり, 達成感にも繋がったと考えている.

最後に, 本プロジェクトで使用したScrapboxについては, 活発な記述によって充実した資料となっており, これは今後も新入生向けの資料として活用できるのではないかと考えられる. これからのRCCでの活動やそれ以外のコーディングにおいて, 本プロジェクトの活動で得られた知見が活きることがあれば幸いである.