プログラミングC演習報告書

第二回レポート課題

【担当教員】　長谷川　亨　教員

【提出者】　城間　大幹　(09B20033)
ソフトウェア科学コース・2年
u381322b@ecs.osaka-u.ac.jp
【提出日】　2021年8月2日

1 課題内容

以下の機能を有するサブセット版のシェルを C 言語を用いて作成せよ。

1. 外部コマンド

   今回自作したコマンド以外の全ての機能。

2. ディレクトリの管理機能

   - cd コマンド
     カレントディレクトリを変更する。ディレクトリ名が指定されなかった場合は、環境変数 HOME に指定されたディレクトリへカレントディレクトリを移動する。
   - pushd コマンド
     ディレクトリスタックへカレントディレクトリを保存する。
   - dirs コマンド
     現在のディレクトリスタックの中身を表示する。
   - popd コマンド
     現在のディレクトリスタックの 1 番上のディレクトリをカレントディレクトリに設定する。

3. ヒストリ機能

   - history コマンド
     これまでに実行したコマンドを、実行した順番とともに表示する。
   - !! コマンド
     !!コマンドによって 1 つ前のコマンドを再度実行する。
   - !string コマンド[1]
     ヒストリに保存されたコマンドの内、string で始まる最新のコマンドを実行する。

4. ワイルド・カード機能

   - *
     コマンドにある ”*” をカレントディレクトリにある全てのファイル名に置換する。

5. プロンプトの変更機能

   - prompt コマンド
     プロンプトを変更する。 文字列が指定されなかった場合は、デフォルトの文字列 ”Command : ” に変更する。

6. スクリプト機能

   1 行に 1 つのコマンドを記述したテキストファイルを標準入力から読み込み、実行する。exit コマンドがなくてもプログラムが終了する。[2]

7. エイリアス機能

[1] string は 1 文字以上の任意の文字列.
[2] コマンドではないことに注意.

- alias コマンド
  コマンドの別名を設定する.。引数を指定しない場合は現在登録されている alias の一覧を表示する。
- unalias コマンド
  unalias コマンドによって、 別名設定されたコマンドを解除する。

8. 自分で考えた機能

   上述の機能以外に、他のシェルなどを参考にして自分で考えた機能を 1 つ追加する。

2 プログラム全体の説明

本章では、プログラム全体の説明を記述する。実装したそれぞれの機能についての説明は、次章以降で機能ごとに説明する。

2.1 シェルの仕様

今回作成したシェルはでは、スクリプト機能についてはプロンプトを表示する前に実行したため、作成したシェルを起動させるにはスクリプト機能に該当する場所をコメントアウトして用いる必要がある。またプロンプトに exit と打ち込むことによりシェルが正常に終了するようになっている。

なお次章以降で各コマンドの詳細な仕様について言及する。

2.2 処理の流れ・実装方法

図 1 に作成したプログラムのフローチャートを示す。

2.2.1 処理の流れ

まず全体的な処理の流れだが、ユーザーからの入力を配列に格納し、そこに格納された文字列と実装したコマンド名の文字列が合致するかを一つずつ確かめる。合致した場合はそのコマンドの機能を実行し、その後に再びユーザーの入力を受け付ける。合致しなかった場合は、外部コマンドを実行する system 関数へと入力された文字列を受け渡して処理し、その後に再びユーザーの入力を受け付ける。この一連の流れを、exit と入力されるまで延々と続ける。

ユーザー入力の処理については、replit 上で配布された simple_shell.c 内にある関数 parse を用いた。該当箇所のコメントが充実しているため詳細な説明は割愛するが、ここではユーザーが入力された文字列を適切に配列に格納するため、空白や改行部分、終端文字の判別を行っていたり、実行グラウンドの選択やシェルの終了などコマンドの状態を判別したりしている。ここでの処理を終えた後 excute_command により実際にコマンドが実行される。

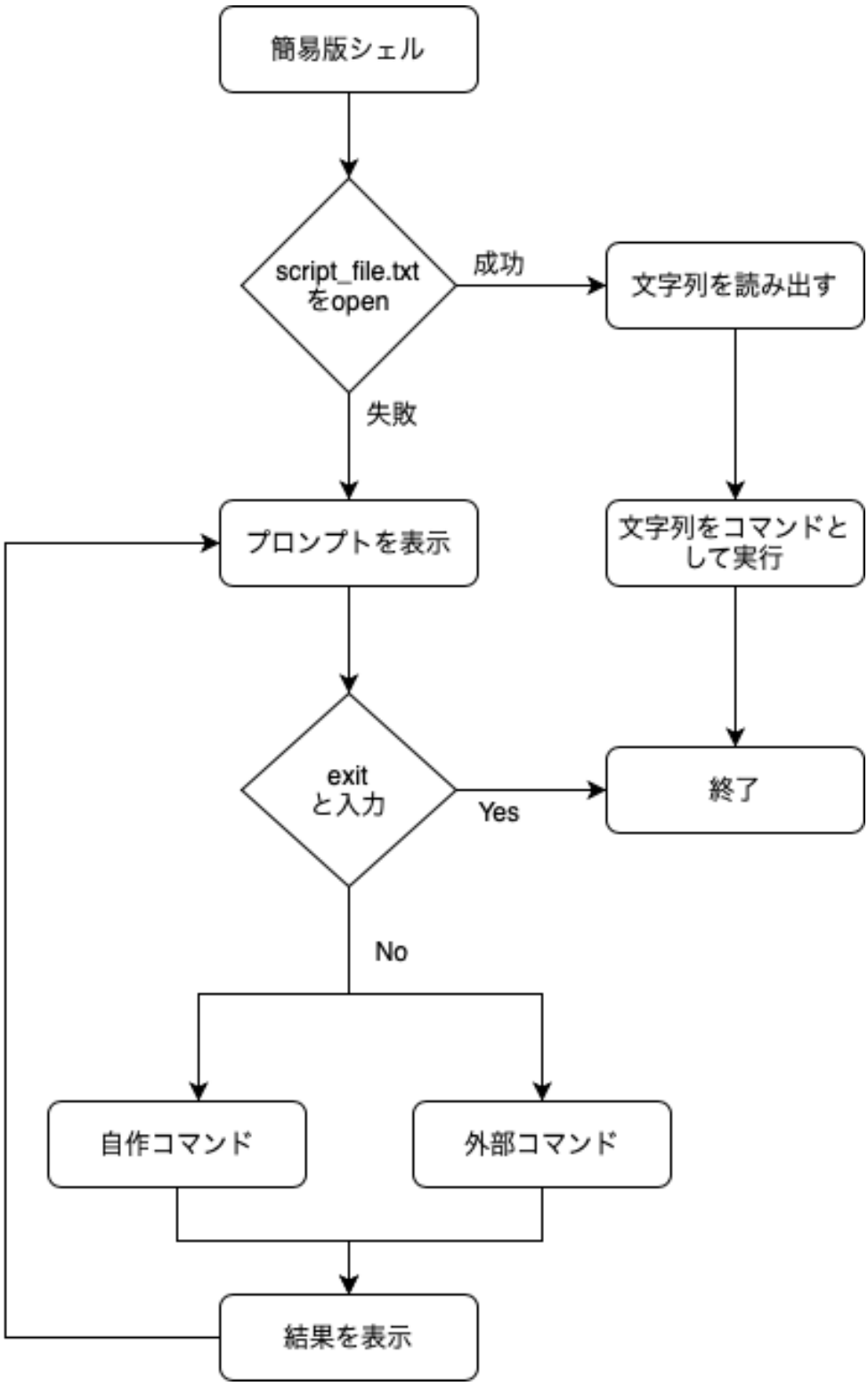

図 1: 作成したプログラムのフローチャート

2.2.2 実装方法

次に実装方法について説明する。今回は replit 上で配布された simple_shell.c に追記するという形で行い、主に関数 excute_command 内に実装するコマンドの処理内容を書いた。またその書き方としては、関数 excute_command に引数として渡された配列 args と実装したコマンドとの文字列比較を if 文を多用して行い、その if 文内に実装するコマンドの詳細な処理や機能を書いた。if 文比較でいずれとも合致しなかった場合に配列 args を system 関数の引数として受け渡し外部コマンドとして実行した。if 文で合致して何らかのコマンドに関連する処理が行われる、あるいは system 関数が実行された場合、関数 excute_command は終了する。

3 外部コマンドの実行機能

本節では外部コマンドの処理、すなわち実装したコマンド以外の処理について説明する。

3.1 仕様

実装したコマンド以外に対してのみ実行できる。また alias コマンドにより別名が付けられた外部コマンドも実行できる。さらにオプションをつけて実行することも可能である。

3.2 処理の流れ

プロンプトに入力された文字列を配列として system 関数の引数として受け渡し実行するのが主となる流れである。しかしながらそれまでの間に、引数の文字列が alias コマンドにより別名が付けられた外部コマンドではないかの判別や、引数に*が含まれていないかを判別するため、場合によっては system 関数に引数として与える文字列を書き換えることもある。

3.3 実装方法

外部コマンドを実行する際の該当する箇所を以下に示す。

```
826       //自作コマンド以外
827       else
828       {
829         pre_register_log(log_root,args);
830         int i=1;
831         char command[100];
832         command[0]='\0';
833         strcpy(command, args[0]);
834         char *ali_command,*temp;
835         ali_command=(char*)malloc(strlen(search_ali(ali_root, args[0]))+1);
836         if(ali_command == NULL)
837         {
838           printf("ERROR\n");
839           exit(-1);
840         }
841         temp=(char*)malloc(strlen(command)+1);
842          if(temp == NULL)
843         {
```

```
844           printf("ERROR\n");
845           exit(-1);
846         }
847         strcpy(ali_command,search_ali(ali_root, args[0]));
848
849           if( strcmp(ali_command,"\0")!=0) strcpy(command, ali_command);
850
851           while (args[i]!=NULL) {
852               strcat(command," ");
853               //ここに*の処理
854               if(strcmp(args[i],"*")==0){
855                   struct stat    filestat;
856                   struct dirent *directory;
857                   DIR           *dp;
858                      dp = opendir(".");
859                         while((directory=readdir(dp))!=NULL){
860                           if(!strcmp(directory->d_name, ".") ||
861                              !strcmp(directory->d_name, ".."))
862                             continue;
863                           if(stat(directory->d_name,&filestat)){
864                             perror("main");
865                             exit(1);
866                           }else{
867                               strcpy(args[i],directory->d_name);
868                               strcpy(temp,command);
869                               strcat(command, args[i]);
870                               system(command);
871                               strcpy(command,temp);
872                           }
873                         }
874                         closedir(dp);
875                   return;
876               }
877               strcat(command, args[i]);
878             if(args[i+1]==NULL)break;
879             i++;
880           }
881           system(command);
882           return;
883     }
```

外部コマンドを実行する上で用いた変数を表 1 に示す。

ここでは独自に作成した pre_register_log 関数、search_ali 関数が用いられているが、pre_register_log 関数については 5.3.1 章で、search_ali 関数については 9.3.1 章で詳細を説明している。

表 1: 外部コマンド実行で用いた変数一覧

変数名	型	用途
i	int 型	args の要素を表す
command	char 型配列	system 関数の引数
ali_command	char 型ポインタ	search_ali 関数の戻り値を格納する
temp	char 型ポインタ	作業用
filestat	構造体	ファイル名を受け取り、そのファイルの inode にある情報を返す。
directory	構造体	ディレクトリエントリを取得する
dp	DIR 型ポインタ	ディレクトリの内容を読み出す

まず基本的な処理として、args の一つ一つの要素を strcat 関数を用いて文字列結合したものを command に格納して

外部コマンドを実行している。

command の中身が別名保存された外部コマンドであるかについての判断だが、この判別には earch_ali 関数を用いて行っておりその戻り値は ali_command に格納している。別名となっていた場合は ali_command に格納された本来のコマンド名を command に格納し、そうでない場合はそのまま command にある文字列を用いている。

*を含む場合の処理については 6.3 章で説明する。

3.4 テスト

3.4.1 テスト方法

仕様や実装方法で説明した箇所が実現されているかを確認する。

3.4.2 テスト結果

実際のテスト結果を以下に示す。

```
\small{
~/report2-shiromadaiki$ gcc simple_shell.c
~/report2-shiromadaiki$ ./a.out
Command : ls
a.out  cp_target  main.c  script_file.txt  simple_shell.c
Command : ls -l
total 52
-rwxr-xr-x 1 runner runner 22384 Jul 29 01:06 a.out
drwxr-xr-x 1 runner runner     0 Jul 29 01:03 cp_target
-rw-r--r-- 1 runner runner     0 Jul  4 05:33 main.c
-rw-r--r-- 1 runner runner    11 Jul 26 15:17 script_file.txt
-rw-r--r-- 1 runner runner 23508 Jul 29 01:01 simple_shell.c
Command : alias sl ls
Command : sl
a.out  cp_target  main.c  script_file.txt  simple_shell.c
Command : sl -l
total 52
-rwxr-xr-x 1 runner runner 22384 Jul 29 01:06 a.out
drwxr-xr-x 1 runner runner     0 Jul 29 01:03 cp_target
-rw-r--r-- 1 runner runner     0 Jul  4 05:33 main.c
-rw-r--r-- 1 runner runner    11 Jul 26 15:17 script_file.txt
-rw-r--r-- 1 runner runner 23508 Jul 29 01:01 simple_shell.c
}
```

この結果より外部コマンドが正しく実行されており、自作した alias コマンドにより sl に別名設定された ls もオプションをつけても実行できるとわかる。以上より前章で言及したような外部コマンド機能は正常に実装されていると言える。

4 ディレクトリの管理機能

4.1 仕様

4.1.1 cd コマンド

引数として指定したディレクトリに移動する。パスは絶対パス、相対パスのいずれも実行可能である。また引数を指定しない場合は、環境変数 HOME に指定されたディレクトリへカレントディレクトリが移動するようになっている。また、移動をわかりやすくするため、cd コマンド実行前後でのパスの状態を出力するようにした。

4.1.2 pushd コマンド

カレントディレクトリのパスを取得し、ディレクトリスタックに保存する。ディレクトリに追加する際はスタックの先頭から追加していく。

4.1.3 dirs コマンド

ディレクトリスタックの中身を”Directory stack = ”の後に続けて出力する。ディレクトリスタックが空である場合は＝の後に何も出力されない。

4.1.4 popd コマンド

ディレクトリスタックの先頭にあるパスを取得しカレントディレクトリをそこに変更する。ディレクトリスタックが空である場合は、その旨を出力する。また popd コマンドを実行した後のパスも出力する。また popd コマンドを実行した際には、ディレクトリスタックの一番上の要素が削除され、元のディレクトリスタックの二番目の要素が一番上の要素となる。

4.2 処理の流れ

4.2.1 cd コマンド

まず cd コマンドを実行する前のファイルパスを出力するために、getcwd 関数によりカレントディレクトリのパスを取得しそれを表示する。次に引数の有無を判別し、引数がある場合は chdir 関数により args[1] に格納されたパスへ移動し、なかった場合は getenv 関数により環境変数 HOME が設定されているところのパスを取得し、そのパスに chdir 関数で移動する。そしてパスの変更が正常に実行されたら、変更後のパスを同様にして取得し出力する。

4.2.2 pushd コマンド

カレントディレクトリのパスを取得し、ディレクトリスタックに保存する。ディレクトリに追加する際はスタックの先頭から追加していく。getcwd 関数によりカレントディレクトリのパスを取得し、そのパスの文字列をディレクトリスタックに追加する。なおディレクトリスタックは単方向リストとなっており、取得したパスをスタックの先頭に保存していく。

4.2.3 dirs コマンド

スタックの先頭を示すアドレスをはじめに参照し、pushd コマンドで保存されたスタックの内容を先頭から末尾まで出力する。

4.2.4 popd コマンド

スタックの先頭のパスを取得し、そのパスに chdir 関数で移動する。そしてパスの変更が正常に実行されたら、変更後のパスを取得し出力する。

4.3 実装方法

4.3.1 cd コマンド

cd コマンドの実装に該当する箇所を以下に示す。

```
591     //cd ここから
592     if(strcmp(args[0],command1)==0||
593        strcmp(search_ali(ali_root, args[0]),command1)==0)
594     {
595         pre_register_log(log_root,args);
596     // カレントディレクトリ取得
597     getcwd(pathname, PATHNAME_SIZE);
598     fprintf(stdout,"変更前のファイルパス:%s\n", pathname);
599
600       if (args[1]==NULL)
601        {
602
603     // カレントディレクトリ変更
604        char* str = getenv( "HOME");
605        if( str == NULL ){
606         fputs( "環境変数の取得に失敗しました。\n", stderr );
607         exit(1);
608        }
609        chdir(str); // チェンジディレクトリ
610        getcwd(str, PATHNAME_SIZE);
611        fprintf(stdout,"現在のファイルパス:%s\n", str);
612            return;
613       }
614       else
615       {
616         // カレントディレクトリ変更
617        if(chdir(args[1])==-1)
618        {
619         printf("No such file or directory\n");
620        }
621        else
622        {
623         getcwd(pathname, PATHNAME_SIZE);
624         fprintf(stdout,"現在のファイルパス:%s\n", pathname);
625        }
626       }
627         strcat(command1," ");
628         strcat(command1,args[1]);
629         return;;
630     }
631       // cd ここまで
```

cd コマンドを実行する上で用いた変数を表 2 に示す。

またここでは独自に作成した pre_register_log 関数、search_ali 関数が用いられているが、pre_register_log 関数については 5.3.1 章で、search_ali 関数については 9.3.1 章で詳細を説明している。

まず getcwd で変更前のパスを取得し fprintf でそこで取得したパスを出力している。

次に args[1] の中身を調べ、args[1] が NULL すなわち引数がないときは、getenv 関数により環境変数 HOME が設定されているところのパスを取得し str に格納し、その str を引数として chdir 関数を実行しカレントディレクトリを変更する。そして再度 getcwd で変更後のパスを取得し fprintf でそこで取得したパスを出力して終える。引数がある場合は、args[1] を chdir 関数の引数として実行してカレントディレクトリを変更する、同様に再度 getcwd で変更後のパスを取得し fprintf でそこで取得したパスを出力して終える。引数として不適切なものが指定された場合は、”No such file or directory” を出力し終える。

表 2: cd コマンドの実装で用いた変数一覧

変数名	型	用途
command1	char 型ポインタ	文字列 cd を示す
pathname	char 型配列	getcwd で取得した文字列を格納する
PATHNAME_SIZE	定数	取得するパスの文字列のサイズ
str	char 型ポインタ	getenv で取得した文字列を格納する

4.3.2 pushd コマンド

pushd のコマンドの実装に該当する箇所を以下に示す。

```
633       //pushd
634     else if(strcmp(args[0],command2)==0||
635             strcmp(search_ali(ali_root, args[0]),command2)==0)
636     {
637         pre_register_log(log_root,args);
638         getcwd(pathname, PATHNAME_SIZE);
639         char *str=pathname;
640         push(stack_root,str);
641     }
642       //pushd ここまで
```

pushd コマンドを実行する上で用いた変数を表 6 に示す。

またここでは独自に作成した pre_register_log 関数、search_ali 関数が用いられているが、pre_register_log 関数については 5.3.1 章で、search_ali 関数については 9.3.1 章で詳細を説明している。

表 3: pushd コマンドの実装で用いた変数一覧

変数名	型	用途
command2	char 型ポインタ	文字列 pushd を示す
pathname	char 型配列	getcwd で取得した文字列を格納する
PATHNAME_SIZE	定数	取得するパスの文字列のサイズ
str	char 型ポインタ	getenv で取得した文字列を格納する

まず getcwd で変更前のパスを取得し str に格納する。次に、str と stack_root を引数として関数 push を実行する。以下関数 push について説明する。

まず関数 push で扱う構造体 stack の該当箇所を以下に、定義を表 4 に示す。

```
15 typedef struct Node{
16   char *data;
17   struct Node *next;
18 }stack;
```

表 4: 構造体 stack の定義

変数名	型	用途
data	char 型ポインタ	パスの文字列を示す
next	stack 型のポインタ	次の構造体 stack のアドレスを示す

表 4 のような構造体を用いてディレクトリスタックを実現した。

次に関数 push の定義の該当箇所を以下に、定義に用いた変数を表 5 に示す。

```
287 void push(stack** stack_root, char* str) {
288   stack* new_stack;
289   new_stack = (stack*)malloc(sizeof(stack));
290     if(new_stack == NULL)
291    {
292     printf("ERROR\n");
293     exit(-1);
294    }
295    new_stack->data =(char*)malloc(strlen(str)+1);
296    if(new_stack->data == NULL)
297    {
298     printf("ERROR\n");
299     exit(-1);
300    }
301    strcpy(new_stack->data, str);
302    new_stack->next = NULL;
303   if(*stack_root==NULL)
304   {
305    *stack_root = new_stack;
306   }
307   else
308   {
309       new_stack->next =*stack_root ;
310       *stack_root = new_stack;
311   }
312   return;
313 }
```

表 5: 関数 push で用いた変数

変数名	型	用途
str	char 型ポインタ	stack に追加するパスの文字列を示す
stack_root	stack 型のポインタのポインタ	stack の先頭アドレスを示すポインタのポインタ
new_stack	stack 型ポインタ	新たに追加する構造体

まず新しく追加する構造体である new_stack を作成し、malloc 関数によりその領域を確保する。new_stack->data についても同様である。次に str の内容を new_stack->data に strcpy 関数によりコピーする。そして最後に new_stack->next のアドレスを指定する。一番目に stack 追加される場合は new_stack->next は当然 NULL である。それ以降 stack に追加する際には、まず new_stack->next のアドレスを追加される前のスタックの先頭のアドレスを指していた*stack_root にし、 *stack_root のアドレスを新たに追加された new_stack のアドレスに更新する。これにより *stack_root は常に stack の先頭アドレスを指すことができる。

また stack_root をポインタのポインタとしているのは pushd コマンドは関数 excute_command 内で実行されているからである、というのもポインタで渡してしまうと値渡しになり関数 excute_command が終了すると元の状態に戻る、すなわち stack の先頭アドレスが更新されないからである。故に参照渡しとなるようにポインタのポインタを用いている。

4.3.3 dirs コマンド

dirs コマンドの実装に該当する箇所を以下に示す。

```
644       //dirs
645      else if(strcmp(args[0],command3)==0||
646              strcmp(search_ali(ali_root, args[0]),command3)==0)
647      {
648         pre_register_log(log_root,args);
649         print_stack(stack_root);
650      }
651      //dirs ここまで
```

dirs コマンドを実行する上で用いた変数を表 6 に示す。

またここでは独自に作成した pre_register_log 関数、search_ali 関数が用いられているが、pre_register_log 関数については 5.3.1 章で、search_ali 関数については 9.3.1 章で詳細を説明している。

表 6: dirs コマンドの実装で用いた変数一覧

変数名	型	用途
command3	char 型ポインタ	文字列 dirs を示す

stack_root を引数として関数 print_stack を実行しディレクトリスタックの中身を表示する。

関数 print_stack の定義の該当箇所を以下に、定義に用いた変数を表 7 に示す。

```
334 void print_stack(stack **stack_root) {
335     stack *selectNode = *stack_root;
336     printf("Directory stack = ");
337     while(selectNode != NULL) {
338         printf("%s",selectNode->data);
339         selectNode = selectNode->next;
340         if( selectNode != NULL ) {
341             printf(", ");
342         }
343     }
344     printf("\n");
345     return;
346 }
```

表 7: 関数 stack で用いた変数

変数名	型	用途
stack_root	stack 型のポインタのポインタ	stack の先頭アドレスを示すポインタのポインタ
selectNode	stack 型ポインタ	作業用

stack の先頭アドレスを指す stack_root を selectNode の初期値として用いる。selectNode->data の中身を出力したあと、selectNode に次の stack のアドレスである selectNode->next を代入し、再度その中身である selectNode->data を示す。この一連の流れを selectNode->next が NULL となるまで続ける。

4.3.4 popd コマンド

popd コマンドの実装に該当する箇所を以下に示す。

```
653      //popd
654      else if(strcmp(args[0],command4)==0||
655              strcmp(search_ali(ali_root, args[0]),command4)==0)
656      {
657       pre_register_log(log_root,args);
658       char *str;
659       str=pop(stack_root);
660          if(str==NULL){
661              printf("ディレクトリスタックが空です\n");
662              return;
663          }
664       chdir(str);
665       getcwd(str, PATHNAME_SIZE);
666       fprintf(stdout,"現在のファイルパス:%s\n", str);
667      }
668      //popd ここまで
```

popd コマンドを実行する上で用いた変数を表 8 に示す。

またここでは独自に作成した pre_register_log 関数、search_ali 関数が用いられているが、pre_register_log 関数については 5.3.1 章で、search_ali 関数については 9.3.1 章で詳細を説明している。

表 8: popd コマンドの実装で用いた変数一覧

変数名	型	用途
command4	char 型ポインタ	文字列 popd を示す
str	char 型ポインタ	関数 pop の戻り値を格納する

stack_root を引数として関数 pop を実行し stack の先頭の data の内容を取得し str に代入する。その後 str を引数として chdir 関数を実行してカレントディレクトリを変更し、変更後のカレントディレクトリを getcwd 関数で取得して出力する。以下関数 pop について説明する。

関数 pop の定義の該当箇所を以下に、定義に用いた変数を表 9 に示す。

```
315 char* pop(stack **stack_root){
316   char *directory;
317   stack *next, *fr;
318     next = (stack*)malloc(sizeof(stack));
319   if(next == NULL){
320     printf("ERROR\n");
321     exit(-1);
322   }
323   if(*stack_root == NULL){
324     return NULL;
325   }
326   directory = (*stack_root) -> data;
327   next = (*stack_root) -> next;
328   fr = *stack_root;
329   *stack_root = next;
330   free(fr);
331   return directory;
332 }
```

表 9: 関数 pop で用いた変数

変数名	型	用途
stack_root	stack 型のポインタのポインタ	stack の先頭アドレスを示すポインタのポインタ
next	stack 型ポインタ	popd コマンド実行後の stack の先頭アドレスに用いる作業用ポインタ
fr	stack 型ポインタ	メモリ解放用ポインタ
directory	char 型ポインタ	戻り値

popd コマンド実行前の stack の先頭アドレスを示す*stack_root を用いて、まず戻り値となる directory に (*stack_root) -> data を格納する。次に*stack_root が指すアドレスを popd コマンド実行前の stack の二番目の構造体のアドレス、すなわち (*stack_root) -> next を指すようにすれば良いので *stack_root に (*stack_root) -> next を代入する。そして最後に、読み出した stack は削除するので、*stack_root がもともと指していたアドレスを表すポインタ、メモリを free により解放する。

4.4 テスト

4.4.1 テスト方法

仕様や実装方法で説明した箇所が実現されているかを確認する。

4.4.2 テスト結果

実際のテスト結果を以下に示す。なお/a.out を実行したディレクトリは/home/runner/report2-shiromadaiki 下で、ここには temp というディレクトリが存在する。

- cd コマンド

```
~/report2-shiromadaiki$ gcc simple_shell.c
~/report2-shiromadaiki$ ./a.out
Command : cd temp
変更前のファイルパス:/home/runner/report2-shiromadaiki
現在のファイルパス:/home/runner/report2-shiromadaiki/temp
Command : cd /home/runner/report2-shiromadaiki
変更前のファイルパス:/home/runner/report2-shiromadaiki/temp
現在のファイルパス:/home/runner/report2-shiromadaiki
Command : cd temp2
変更前のファイルパス:/home/runner/report2-shiromadaiki
No such file or directory
Command : cd temp
変更前のファイルパス:/home/runner/report2-shiromadaiki
現在のファイルパス:/home/runner/report2-shiromadaiki/temp
Command : cd
変更前のファイルパス:/home/runner/report2-shiromadaiki/temp
現在のファイルパス:/home/runner
Command : cd report2-shiromadaiki
変更前のファイルパス:/home/runner
現在のファイルパス:/home/runner/report2-shiromadaiki
Command : cd ..
変更前のファイルパス:/home/runner/report2-shiromadaiki
現在のファイルパス:/home/runner
```

上の結果の 1 番目と 2 番目より、パス指定が絶対パス、相対パスいずれの場合でもカレントディレクトリが変更されていることがわかる。

また上の結果の 3 番目より存在しないディレクトリをパスと指定した場合は正常にエラーメッセージ”No such file or directory”が出力されている。

さらに上の結果の 4 番目より引数がない場合はホームディレクトリへカレントディレクトリが変更されていることがわかる。

そして最後に、上の結果の 7 番目から、引数として.. を指定するとカレントディレクトリの一つ上のディレクトリに移動していることがわかる。

以上より、実装した cd コマンドは正しく実装されていると言える。

- pushd, dirs, popd コマンド

```
~/report2-shiromadaiki$ gcc simple_shell.c
~/report2-shiromadaiki$ ./a.out
Command : pwd
/home/runner/report2-shiromadaiki
Command : pushd
Command : dirs
Directory stack = /home/runner/report2-shiromadaiki
Command : cd
変更前のファイルパス:/home/runner/report2-shiromadaiki
現在のファイルパス:/home/runner
Command : pwd
/home/runner
Command : pushd
Command : dirs
Directory stack = /home/runner, /home/runner/report2-shiromadaiki
Command : cd report2-shiromadaiki
変更前のファイルパス:/home/runner
現在のファイルパス:/home/runner/report2-shiromadaiki
Command : popd
現在のファイルパス:/home/runner
```

```
Command : dirs
Directory stack = /home/runner/report2-shiromadaiki
Command : pwd
/home/runner
Command : popd
現在のファイルパス:/home/runner/report2-shiromadaiki
Command : pwd
/home/runner/report2-shiromadaiki
Command : dirs
Directory stack =
Command : popd
ディレクトリスタックが空です
```

上の結果から pushd コマンドによりディレクトリスタックが適切に作成されていることがわかる。というのも pwd で出力したカレントディレクトリ名が pushd コマンドを実行するとスタックに格納されているからである。

また後者の pushd を実行した後のディレクトリスタックの中身、popd を実行したあとの dirs の結果の双方を見ると、ディレクトリスタックには新しい要素は先頭に追加し、読み出す時も先頭から読み出すということが実現できていることがわかる。

なおディレクトリスタックが空の状況で dirs、popd を実行してみた結果を見ても、その状況を正しく処理できているとわかる。

以上の結果より、pushd、dirs、popd コマンドは適切に実装できていると言える。

5 ヒストリー機能

5.1 仕様

5.1.1 history コマンド

これまでに実行したコマンドの一覧を実行した順番とともに示す。なお history コマンドの一個前までのコマンドを示す。またヒストリーに記録できるのは 32 個のコマンドであり、33 個目以降のコマンドはヒストリーに記録されないようにした。

5.1.2 !系コマンド

!!とプロンプトに打ち込んだ場合は、その直前に実行したコマンドを再度実行する。ヒストリーに何も記録されていない場合に!!を実行した場合はエラーメッセージを出力する。また!string[3]と打ち込んだ場合は、string に該当する文字列から始まるコマンドのうち、最も新しく実行したものを再度実行する。string が 0 文字だった場合は!string コマンドの使用方法を、該当するコマンドが見当たらなかった場合はエラーメッセージを出力するようにした。

5.2 処理の流れ

5.2.1 history コマンド

実行したコマンドを順に記録した単方向リストを先頭から末尾まで読み出し、その内容を表示する。

[3]string は一文字以上の任意の文字列

5.2.2 !系コマンド

まず!の後に!が続くかどうかを判別する。!!の場合、実行したコマンドを順に記録した単方向リストから最新のコマンド名を取得し、その文字列と実装したコマンドの文字列が合致するかを一つずつ確かめるという最初に行った流れを再度行う。!string の場合、string の文字列から始まる最新のコマンドを単方向リスト内を検索し、該当するコマンドがあればそのコマンド名を取得し、同様にその文字列と実装したコマンドの文字列が合致するかを一つずつ確かめるという最初に行った流れを再度行う。該当するコマンドがなかったり、書き方が間違っていたりする場合はその旨を出力しプロンプトの入力待機画面に戻る。

5.3 実装方法

5.3.1 history コマンド

history コマンドの実装に該当する箇所を以下に示す。

```
670      //histotry
671      else if(strcmp(args[0],command5)==0||
672              strcmp(search_ali(ali_root, args[0]),command5)==0){
673         print_log(log_root);
674         register_log(log_root,command5);
675      }
```

history コマンドを実行する上で用いた変数を表 10 に示す。

ここでは独自に作成した search_ali 関数が用いられているがこの関数については 9.3.1 章で詳細を説明している。

表 10: history コマンドの実装で用いた変数一覧

変数名	型	用途
command5	char 型ポインタ	文字列 history を示す

log_root を引数として関数 print_log を実行し、実行したコマンドを順番ともに表示する。以下ここで用いた関数 pre_register_log、register_log、print_log とヒストリー機能の実装に伴い扱う構造体 histor について説明する。

まずヒストリー機能の実装に伴い扱う構造体 history の定義の該当箇所を以下に、詳細を表 11 に示す。

```
23 typedef struct command_log{
24   char *data;
25   struct command_log *next;
26 }hisotry;
```

表 11: 構造体 history の定義

変数名	型	用途
data	char 型ポインタ	実行したコマンドの文字列を示す
next	history 型のポインタ	次の構造体 history のアドレスを示す

表 11 のような構造体を用いて実行したコマンドを順に記録するヒストリー機能を実現した。そして以下に構造体 history を用いる関数について述べる。

関数 pre_register_log の定義に該当する箇所を以下に、定義に用いた変数を表 12 に示す。

```
348 void pre_register_log(hisotry** log_root,char* args[]){
349     int i=1;
350     char command[100];
351     command[0]='\0';
```

```
352     strcpy(command, args[0]);
353     while (args[i]!=NULL) {
354           strcat(command," ");
355           strcat(command, args[i]);
356         if(args[i+1]==NULL)break;
357         i++;
358       }
359       register_log(log_root,command);
360 }
```

表 12: 関数 pre_register_log で用いた変数

変数名	型	用途
log_root	history 型のポインタのポインタ	history の先頭アドレスを示すポインタのポインタ
args	char 型配列ポインタ	ユーザーが入力した文字列
int	int 型	args の添字
command	char 型配列	関数 register_log への引数

ここでは args の要素を全て結合し、command に一つの文字列として受け渡している。多くの場合、各コマンドを実行する前にこの関数を実行し、history に追加している。

次に関数 register_log の定義に該当する箇所を以下に、定義に用いた変数を表 13 に示す。

```
362 void register_log(hisotry** log_root, char* str) {
363   hisotry* new_history;
364   hisotry* selectNode = *log_root;
365   new_history = (hisotry*)malloc(sizeof(hisotry));
366     if(new_history == NULL)
367    {
368     printf("ERROR\n");
369     exit(-1);
370    }
371    new_history->data =(char*)malloc(strlen(str)+1);
372    if(new_history->data == NULL)
373    {
374     printf("ERROR\n");
375     exit(-1);
376    }
377    strcpy(new_history->data, str);
378    new_history->next = NULL;
379
380   if(*log_root==NULL)
381   {
382    *log_root = new_history;
383   }
384
385   else
386   {
387       int count=1;
388       while (selectNode->next != NULL) {
389             count++;
390             selectNode = selectNode->next;
391         }
392         if(count==32){
393           printf("history の数が上限です\n");
394         }
395         else{
396       selectNode->next = new_history;
397       new_history->next =NULL ;
```

```
398         }
399    }
400    return;
401 }
```

表 13: 関数 register_log で用いた変数

変数名	型	用途
log_root	history 型のポインタのポインタ	history の先頭アドレスを示すポインタのポインタ
str	char 型ポインタ	ヒストリーに記録する文字列
count	int 型	ヒストリーに記録されているコマンドの数をカウントする
new_history	history 型ポインタ	新たに追加する構造体
selectNode	history 型ポインタ	作業用

ここではまず新しく追加する構造体である new_history を作成し、malloc 関数によりその領域を確保する。new_history->data についても同様である。次に str の内容を new_history->data に strcpy 関数によりコピーする。また新しく作成した構造体は history の末尾に追加するので selectNode が NULL になるまで while 文を回し、NULL になった時に追加前の history 末尾のポインタ next に new_history のアドレスを渡し、new_history->next を NULL にする。

また log_root をポインタのポインタとしているのは関数 register_log は関数 excute_command 内で実行されているからである、というのもポインタで渡してしまうと値渡しになり関数 excute_command が終了すると元の状態に戻る、history に追加した要素が関数 excute_command 終了時に保存されないからである。故に参照渡しとなるようにポインタのポインタを用いている。

次に関数 print_log の定義に該当する箇所を以下に、定義に用いた変数を表 14 に示す。

```
403 void print_log(hisotry **log_root) {
404     hisotry *selectNode = *log_root;
405     int i=1;
406     while(selectNode != NULL) {
407         printf("%d %s\n",i,selectNode->data);
408         selectNode = selectNode->next;
409         i++;
410     }
411     return;
412 }
```

表 14: 関数 print_log で用いた変数

変数名	型	用途
log_root	history 型のポインタのポインタ	history の先頭アドレスを示すポインタのポインタ
selectNode	history 型ポインタ	作業用
i	int 型	実行した順番を示す

history の先頭アドレスを指す log_root を selectNode の初期値として用いる。実行したコマンドの順番と selectNode->data の中身を出力したあと、selectNode に次の history のアドレスである selectNode->next を代入し、再度その中身である selectNode->data を示す。この一連の流れを selectNode->next が NULL となるまで続ける。

5.3.2 !系コマンド

!系のコマンドの実装に該当する箇所を以下に示す。

```
676     // !系コマンド
677
678      else if (args[0][0]=='!'){
679        if (args[0][1]=='!'){
680            if( latest_log(log_root)==NULL) return;
681         strcpy(args[0],latest_log(log_root));
682         int j=0;
683          char *p1,*p2;
684          p1 =(char*)malloc(strlen(args[0])+1);
685          if(p1 == NULL)
686           {
687               printf("ERROR\n");
688               exit(-1);
689           }
690          strcpy(p1,args[0]);
691          p2 = strtok(p1, " ");
692          while (p2)
693                  {
694                      if(j!=0){
695                          args[j] =(char*)malloc(strlen(args[0])+1);
696                          if(args[j] == NULL)
697                               {
698                               printf("ERROR\n");
699                               exit(-1);
700                               }
701                      }
702                          strcpy(args[j],p2);
703                          p2 = strtok(NULL, " ");
704                      j++;
705                  }
706         goto hisotry;
707         }
708
709        if (*(args[0]+1)=='\0'){
710          printf("!の後に文字を続けてください\n");
711          return;
712         }
713         int i=1;
714          char tmp[10];
715         while(1) {
716             if(i==1){
717                 strcpy(tmp,&args[0][i]);
718                 i++;}
719             else{
720             strcat(tmp,&args[0][i]);
721              i++;
722             if (*(args[0]+i)=='\0') break;
723           }
724         }
725
726          if( search_log(log_root,tmp)==NULL) {
727              printf("該当するコマンドがありません\n");
728              return;}
729            strcpy(args[0],search_log(log_root,tmp));
730          //複数行ある場合 args[0] に全て入ってしまっている
731          int j=0;
732          char *p1,*p2;
733          p1 =(char*)malloc(strlen(args[0])+1);
734          if(p1 == NULL)
735           {
736               printf("ERROR\n");
737               exit(-1);
```

```
738           }
739          strcpy(p1,args[0]);
740          p2 = strtok(p1, " ");
741          while (p2)
742                  {
743                      if(j!=0){
744                          args[j] =(char*)malloc(strlen(args[0])+1);
745                          if(args[j] == NULL)
746                               {
747                               printf("ERROR\n");
748                               exit(-1);
749                               }
750                      }
751                          strcpy(args[j],p2);
752                          p2 = strtok(NULL, " ");
753                      j++;
754                  }
755             goto hisotry;
756       }
757      //!系コマンドここまで
```

!系のコマンドを実行する上で用いた変数を表 15 に示す。

表 15: !系コマンドの実装で用いた変数一覧

変数名	型	用途
i	int 型	配列 args の添字
temp	char 型配列	!string コマンドの string にあたる文字列を格納する
j	int 型	ポインタ args の要素をさす
p1	char 型ポインタ	作業用
p2	char 型ポインタ	分割した文字列を格納

args[0][1] の要素で場合分けする。args[0][1] が!であれば、log_root を引数として関数 latest_log を実行し、history の末尾の構造体からコマンド名を取得し、args[0] の要素を書き換える。ここで注意することとして、関数 latest_log の戻り値が空白を含む文字列だった場合、args[0] にその文字列が入ることになるので、strtok 関数により空白で文字列を分割し、args[1] 以降の要素に代入する。実行するコマンドの準備が整ったら、goto によりラベル history に飛び、再度 args とコマンド名の比較を行い、該当する箇所でコマンドを実行する。

次に関数 latest_log の定義に該当する箇所を以下に、定義に用いた変数を表 16 に示す。

```
414 char* latest_log(hisotry **log_root) {
415     hisotry *selectNode = *log_root;
416     if(selectNode==NULL){
417         printf("history が空です\n");
418         return NULL;
419     }
420     hisotry *p = NULL;
421     while(selectNode != NULL) {
422         p=selectNode;
423         selectNode = selectNode->next;
424     }
425     return p->data;
426 }
```

history の先頭アドレスを指す log_root を selectNode の初期値として用いる。selectNode に次の history のアドレスである selectNode->next を代入するというループを selectNode->next が NULL になるまで続ける。ループを抜けた地点が history の末尾を示すので、その構造体からコマンド名を取得して終える。history が空である場合はエラーメッセージを出力し終える。

表 16: 関数 latest_log で用いた変数

変数名	型	用途
log_root	history 型のポインタのポインタ	history の先頭アドレスを示すポインタのポインタ
selectNode	history 型ポインタ	作業用
p	history 型ポインタ	作業用

次に args[0][1] が!でないときを考える。string にあたる文字列、すなわち args[0][1] 以降の要素を取り出し temp に格納する。そして log_root を引数として関数 search_log を実行し、history 内に string から始まり、かつ最も直近に実行されたものがあればそのコマンド名を取得し、args[0] の要素を書き換える。ここで注意することとして、関数 latest_log の戻り値が空白を含む文字列だった場合、args[0] にその文字列が入ることになるので、strtok 関数により空白で文字列を分割し、args[1] 以降の要素に代入する。実行するコマンドの準備が整ったら、goto によりラベル history に飛び、再度 args とコマンド名の比較を行い、該当する箇所でコマンドを実行する。

次に関数 search_log の定義に該当する箇所を以下に、定義に用いた変数を表 17 に示す。

```
427 char* search_log(hisotry **log_root,char*name) {
428     hisotry *selectNode = *log_root;
429     hisotry *p = NULL;
430     char *str = NULL;
431     while(selectNode != NULL) {
432         p=selectNode;
433         if(p->data[0]==*(name+0)){
434             if(strstr(p->data,name)!=NULL){
435             str=strstr(p->data,name);
436             }}
437         selectNode = selectNode->next;
438         if (p==NULL)return NULL;
439     }
440     return str;
441 }
```

表 17: 関数 search_log で用いた変数

変数名	型	用途
log_root	history 型のポインタのポインタ	history の先頭アドレスを示すポインタのポインタ
selectNode	history 型ポインタ	作業用
p	history 型ポインタ	作業用
name	char 型ポインタ	!string の string
str	char 型ポインタ	戻り値を格納する

history の先頭アドレスを指す log_root を selectNode の初期値として用い、selectNode->next が NULL になるまでループを続ける。したがって、history 内に string を含むコマンド名が複数あった場合は、もっとも直近に実行されたものが str に格納されるようになっている。またコマンド名検索については strstr 関数を用いるが、strstr 関数の仕様として探索対象となる文字列の途中に string を含む場合に関してもマッチしてとみなしてしまう。これは各コマンド名の一文字目と string の一文字目が合致する場合のみ strstr 関数を用いることで防いでいる。このようにして、該当するコマンドが history にあればその文字列を、なければ NULL を返して終える。

なお!!や!!string 自体はコマンドとして history に記録せず、それらにより実行されたコマンドを history に記録している。

5.4 テスト

5.4.1 テスト方法

仕様や実装方法で説明した箇所が実現されているかを確認する。

5.4.2 テスト結果

実際のテスト結果を以下に示す。

```
~/report2-shiromadaiki$ gcc simple_shell.c
~/report2-shiromadaiki$ ./a.out
Command : !!
history が空です
Command : !aaa
該当するコマンドがありません
Command : alias sl ls
Command : sl -l
total 52
-rwxr-xr-x 1 runner runner 22384 Jul 31 00:55 a.out
-rw-r--r-- 1 runner runner    11 Jul 26 15:17 script_file.txt
-rw-r--r-- 1 runner runner 23983 Jul 30 01:15 simple_shell.c
drwxr-xr-x 1 runner runner     0 Jul 29 06:28 temp
Command : !!
total 52
-rwxr-xr-x 1 runner runner 22384 Jul 31 00:55 a.out
-rw-r--r-- 1 runner runner    11 Jul 26 15:17 script_file.txt
-rw-r--r-- 1 runner runner 23983 Jul 30 01:15 simple_shell.c
drwxr-xr-x 1 runner runner     0 Jul 29 06:28 temp
Command : history
1 alias sl ls
2 sl -l
3 sl -l
Command : !
!の後に文字を続けてください

Command : history
1 alias ss date
2 ss
3 alias sl ls
4 sl -l
5 history
6 cd
7 pushd
Command : !s
total 4
drwxr-xr-x 1 runner runner  76 Jul 31 00:55 report2-shiromadaiki
-rw-r--r-- 1 root   root   579 Jan  1  1970 _test_runner.py
            .
            .
            .
Command : alias
ss date
sl ls
Command : alias s
history の数が上限です
alias 引数 1 引数 2 と書いてください
Command : history
1 alias ss date
```

```
2 ss
3 alias sl ls
4 sl -l
5 history
6 cd
7 pushd
8 sl -l
9 pwd
10 who
11 which
12 popd
13 dirs
14 ls
15 ls
16 ls
17 ls
18 ls
19 ls
20 history
21 aa
22 ii
23 uu
24 ee
25 oo
26 hisotry
27 history
28 cd
29 popd
30 popd
31 popd
32 alias
history の数が上限です
Command : pwd
history の数が上限です
/home/runner
Command : history
1 alias ss date
2 ss
3 alias sl ls
4 sl -l
5 history
6 cd
7 pushd
8 sl -l
9 pwd
10 who
11 which
12 popd
13 dirs
14 ls
15 ls
16 ls
17 ls
18 ls
19 ls
20 history
21 aa
22 ii
23 uu
24 ee
25 oo
26 hisotry
```

```
27 history
28 cd
29 popd
30 popd
31 popd
32 alias
history の数が上限です
```

一番目の結果の 1.2 番目より history に何も記録されていない場合に!!や!string コマンドを実行した場合はエラーメッセージが出ていることがわかる。

一番目の結果の 5 番目より!!を入力すると直前に実行したコマンドを再度実行できており、また alias により別名設定がされていたり、オプションがついていたりする場合も問題なく実行されていることがわかる。

一番目の結果の 6 番目より history には!!自体が記録されるのではなく、実際に実行したコマンドが記録されていることがわかる。

一番目の結果の 7 番目より!string のコマンドで string が 0 文字であった場合はエラーメッセージが正しく出力されていることがわかる。

次に二番目の結果を見てみる。!s を入力すると、s から始まり最も直近に実行された sl -l が実行されている。このことから、alias により別名設定がされていたり、オプションがついていたりする場合も問題なく実行されていることがわかり、また stiring から始めるコマンドが複数あった場合でも、最も直近に実行された判別し実行できると言える。

またコマンドを 32 個実行した場合は、”hisotry の数が上限です”というメッセージが出力され、それ以降コマンドを実行しても history に新たにリストが追加されることはないことがわかる。

以上より、実装した!!と!string コマンドは正しく実装されていると言える。

6 ワイルドカード機能

6.1 仕様

カレントディレクトリ内にあるファイルやディレクトリを置換する。置換するファイルについては ls コマンドにより表示されるものを対象とする。

6.2 処理の流れ

args の要素に*を含む場合、opendir 関数でディレクトリを開き、readdir 関数によりファイルを一つずつ読み出す。そこで取得したファイル名を args の要素として置き換え、元のコマンドの引数として実行する。*により置換されるファイルの数の分だけコマンドを実行することになる。

6.3 実装方法

ワイルドカード機能の実装に該当する箇所を以下に示す。

```
853                 //ここに*の処理
854                 if(strcmp(args[i],"*")==0){
855                     struct stat    filestat;
856                     struct dirent *directory;
857                     DIR           *dp;
858                        dp = opendir(".");
859                           while((directory=readdir(dp))!=NULL){
860                             if(!strcmp(directory->d_name, ".") ||
861                                !strcmp(directory->d_name, ".."))
862                               continue;
863                             if(stat(directory->d_name,&filestat)){
864                               perror("main");
```

```
865                             exit(1);
866                           }else{
867                               strcpy(args[i],directory->d_name);
868                               strcpy(temp,command);
869                               strcat(command, args[i]);
870                               system(command);
871                               strcpy(command,temp);
872                           }
873                       }
874                       closedir(dp);
875                   return;
876               }
```

ワイルドカード機能を実装する上で用いた変数を表 18 に示す。

表 18: ワイルドカード機能の実装で用いた変数一覧

変数名	型	用途
temp	char 型ポインタ	作業用
filestat	構造体	ファイル名を受け取り、そのファイルの inode にある情報を返す。
directory	構造体	ディレクトリエントリを取得する
dp	DIR 型ポインタ	ディレクトリの内容を読み出す

opendir 関数によりディレクトリを開く、次に readdir 関数によりディレクトリ項目を読み出し、これを NULL が返されるまで行うことでカレントディレクトリ内の全てのファイルを読み出す。また [1] より各ディレクトリの”.”であるエントリやその親である”..”スキップするべきなので strcmp により判別する。その次に、stat によりファイル名を受け取り、そのファイルの inode にある全ての情報を返す。そしてそこで得たファイル名を取得しそこで得たファイス名をコマンド名を strcat 関数により結合し、コマンドを実行する。このコマンドを実行する流れを readdir 関数で読み出されるファイルの数の分だけ行う。

6.4 テスト

6.4.1 テスト方法

仕様や実装方法で説明した箇所が実現されているかを確認する。

6.4.2 テスト結果

実際のテスト結果を以下に示す。

```
~/report2-shiromadaiki$ gcc simple_shell.c
~/report2-shiromadaiki$ ./a.out
Command : ls
a.out  script_file.txt  simple_shell.c  temp
Command : cat *
ls
date
pwdcat: temp: Is a directory
#include <stdio.h>
#include <ctype.h>
#include <stdlib.h>
#include <dirent.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
```

```
#include <sys/wait.h>
#include <string.h>
          .
          .
          .
```

pwd までが script_file.txt の内容で#include からが simple_shell.c の内容であり、ここでは長くなるため#include <string.h> 以降は省略している。[4]この結果よりカレントディレクトリ内の script_txt, simple_shell.c の内容が cat コマンドにより正しく出力されていることからファイル名置換が正しく実装されているとわかる。また cat コマンドではディレクトリは開けないが、エラーメッセージが正しく出力されていることからディレクトリである temp も*で置換されていることがわかる。

以上の結果よりワイルドカードは正しく実装されていると言える。

7 プロンプト機能

7.1 仕様

プロンプトの入力待機画面で表示される Command という文字列を引数として指定した文字列に変更する。引数を指定しなかった場合はデフォルトの Command 文字列にする。

7.2 処理の流れ

プロンプトで表示する文字列を示す変数の中身を引数の文字列に書き換える。引数がない場合は、初期設定に戻す。

7.3 実装方法

prompt コマンドの実装に該当する箇所を以下に示す。

```
759     //prompt
760      else if (strcmp(args[0],command6)==0||
761               trcmp(search_ali(ali_root, args[0]),command6)==0){
762          pre_register_log(log_root,args);
763          if (args[1]!=NULL){
764              char *temp=args[1];
765              strcpy(*prompt_name,temp);
766              return;
767          }
768          else{
769              char original_name[]="Command";
770              char *init_name=original_name;
771              strcpy(*prompt_name,init_name);
772              return;
773          }
774         //prompt ここまで
```

prompt コマンドを実装する上で用いた変数を表 19 に示す。

またここでは独自に作成した pre_register_log 関数、search_ali 関数が用いられているが、pre_register_log 関数については 5.3.1 章で、search_ali 関数については 9.3.1 章で詳細を説明している。

まず args[1] に要素があるかないかを判断する。args[1] に文字列がある場合は、それが Command に代わり表示する文字列になるので strcpy 関数により*prompt_name に args[1] の文字列をコピーする。args[1] が NULL の時は、デフォルトである文字列 Command を strcpy 関数により*prompt_name にコピーする

[4]a.out の中身も実際は出力されている。

表 19: prompt コマンドの実装で用いた変数一覧

変数名	型	用途
command6	char 型ポインタ	文字列 prompt を示す
temp	char 型ポインタ	作業用
prompt_name	char 型ポインタのポインタ	プロンプトの文字列を表す
original_name	char 型配列	プロンプトの初期設定の文字列 Command を表す
init_name	char 型ポインタ	strcpy 関数に用いる作業用

また prompt_name をポインタのポインタとしているのは関数 register_log は関数 excute_command 内で実行されているからである、というのもポインタで渡してしまうと値渡しになり関数 excute_command が終了すると元の状態に戻る、history に追加した要素が関数 excute_command 終了時に保存されないからである。故に参照渡しとなるようにポインタのポインタを用いている。

7.4 テスト

7.4.1 テスト方法

仕様や実装方法で説明した箇所が実現されているかを確認する。

7.4.2 テスト結果

実際のテスト結果を以下に示す。

```
~/report2-shiromadaiki$ gcc simple_shell.c
~/report2-shiromadaiki$ ./a.out
Command : prompt abc
abc : prompt def
def : prompt
Command :
```

1、2 番目の結果より、prompt コマンドの引数がプロンプト入力待機画面のデフォルト表示の”Command”という文字列に代わって表示されていることがわかる。また 3 番目の結果より、prompt コマンドの引数が指定されていない場合は、デフォルト表示の”Command”という文字列に戻っている。

以上の結果より prompt コマンドは正しく実装されていると言える。

8 スクリプト機能

8.1 仕様

サブセット版のシェルを実行する C ファイルである simple_shell.c と同じディレクトリ内にあるテキストファイルである script_file.txt にある文字列を読み込み、コマンドとして実行する。スクリプト機能を実行した後はプロンプトを表示することなく終了する。また、script_file.txt では、コマンドは空白文字や改行文字により区切られたものを一つとして考えているので、引数などを指定することは想定していない。またここで実行するコマンドは、自作したコマンドではなく外部コマンドとなっている。

8.2 処理の流れ

script_file.txt を読み込み、文字列を先頭から切り出していく。切り出した文字列を格納し、それをコマンドとして実行する。

8.3 実装方法

スクリプト機能の実装に該当する箇所を以下に示す。

```
65     //スクリプト機能
66
67       /*  fp=fopen("script_file.txt","r");
68         if(fp == NULL){
69                  fprintf(stderr, "script_file.txt is not found.\n");
70             }
71         else{
72         cc = getc(fp);                                  // ファイルからの 1 文字読みだし
73
74            while(cc != EOF){
75
76                while(isspace(cc))  cc = getc(fp);
77
78                if(cc == EOF) break;
79
80                tmp = string;
81                while(!(isspace(cc) || cc == EOF)){
82                   if((tmp - string) >= (MAXLENGTH - 1)){
83                        fprintf(stderr, "Too long sentence.\n");
84                        fclose(fp);
85                        exit(1);
86                    }
87                    *tmp++ = cc;
88                    cc = getc(fp);
89                }
90                *tmp = '\0';
91                system(string);
92                }
93         fclose(fp);
94             return 0;
95         }*/
96     //スクリプト機能ここまで
```

スクリプト機能を実装する上で用いた変数を表 20 に示す。

表 20: スクリプトの実装で用いた変数一覧

変数名	型	用途
cc	int 型	getc 関数により読み込む文字を格納する
fp	FILE 型ポインタ	script_file.txt を開くのに用いる
temp	char 型ポインタ	切り出した文字列を格納する
string	char 型配列	temp の大きさ
MAXLENGTH	定数	256

fopen 関数により、script_file.txt を読み取りモードで開き、getc 関数により script_file.txt の内容を 1 文字ずつ読んでいく。これを EOF が現れるまで続ける。空白文字や改行文字で区切られた文字列を一つのコマンドとして用いるため、isspace 関数が正の値を返すまでの連続した文字をポインタ temp に格納していく。そして出来上がった一つの文字列を示す temp を system 関数の引数として、コマンドを実行する。これを EOF が現れるまで切り出した文字列の数の分だけ繰り返す。

8.4 テスト

8.4.1 テスト方法

仕様や実装方法で説明した箇所が実現されているかを確認する。

8.4.2 テスト結果

まず script_file.txt の内容を下記に示す。

```
ls date
pwd
cd
pwd
cd temp
pwd
}
```

次に実際のテスト結果を以下に示す。なお simple_shell.c と同じディレクトリ内に temp というディレクトリが存在する。
\small{
\begin{verbatim}

```
~/report2-shiromadaiki$ gcc simple_shell.c
~/report2-shiromadaiki$ ./a.out
a.out  main  script_file.txt  simple_shell.c  temp
Sun Aug  1 03:20:43 UTC 2021
/home/runner/report2-shiromadaiki
/home/runner/report2-shiromadaiki
sh: 1: temp: not found
/home/runner/report2-shiromadaiki
```

1、2 番目の結果より、script_file.txt から文字列を切り出しコマンドとして実行できていることがわかる。また同結果から空白文字で区切られた文字を一つのコマンドとして認識していることもわかる。

次に残りの結果から、引数を指定したコマンドは正しく実装できていないとわかる。cd コマンドにより、temp に移動することができておらず、temp をコマンドとして認識してしまい、cd 実行後のパスの情報が変わっていないことがわかる。

また実行した cd コマンドは実装した cd コマンドのように変更前、変更後のファイルパスが表示されていないため外部コマンドがここでは実行されていることもわかる。

以上の結果よりスクリプト機能は仕様を満たしていると言える。

9 エイリアス機能

9.1 仕様

9.1.1 alias コマンド

第一引数に設定したい名前を、第二引数に別名設定したいコマンド名指定する。別名設定するコマンドが、仮に存在しないコマンドであっても別名に設定される。またオプションがついたままの形でコマンドを別名設定することはできない。一つのコマンドに複数の別名設定が可能である。複数のコマンドに同一の別名を設定する場合は想定していない。

引数が指定されなかった場合は、alias コマンドにより別名設定されているコマンドとその別名の一覧を表示する。

第二引数がない場合はエラーメッセージが出る。

9.1.2 unalias コマンド

引数に別名設定されたコマンドの別名を指定すると、そのコマンドの別名設定が解除される。引数として指定された別名のみを解除するのであり、それ以外に別名設定されていたとしてもその別名は解除しない。

引数が指定されていない場合はエラーメッセージが出る。

9.2 処理の流れ

9.2.1 alias コマンド

エイリアスを設定したコマンドとその別名を保存した単方向リストを作成して用いる。引数がない場合は、単方向リストの内容を出力する。引数がある場合は、新たにコマンドをリストに登録する。

9.2.2 unalias コマンド

リスト内を探索し、引数と合致する別名があればその別名を元のコマンド名に戻して解除する。

9.3 実装方法

9.3.1 alias コマンド

alias コマンドの実装に該当する箇所を以下に示す。

```
776      //alias
777      else if (strcmp(args[0],command7)==0||
778               strcmp(search_ali(ali_root, args[0]),command7)==0){
779          pre_register_log(log_root,args);
780          if(args[1]==NULL){
781              print_ali(ali_root);
782              return;
783          }
784          else {
785              if(args[2]==NULL) {
786                  printf("alias 引数 1 引数 2 と書いてください\n");
787                  return;
788              }
789              register_ali(ali_root,args[1],args[2]);
790              return;
791          }
792      }
793     //alias ここまで
```

alias コマンドを実行する上で用いた変数を表 21 に示す。

表 21: alias コマンドの実装で用いた変数一覧

変数名	型	用途
command7	char 型ポインタ	文字列 alias を示す

またここでは独自に作成した pre_register_log 関数が用いられているが、その説明については 5.3.1 章にある。

次に alias コマンドの実装に用いた関数、register_ali、search_ali、print_ali が使用する構造体 ali に該当する箇所を以下に、定義する上で用いた変数を表 22 に示す。

```
33 typedef struct alias{
34   char *original;
35   char *ali_name;
36   struct alias *next;
37 }ali;
```

表 22: 構造体 ali の定義

変数名	型	用途
original	char 型ポインタ	コマンドのもともとの名前を示す
ali_name	char 型ポインタ	コマンドの別名を示す
next	history 型のポインタ	次の構造体 ali のアドレスを示す

表 22 のような構造体を用いて alias コマンドを実装した。そして以下に構造体 ali を用いる関数、register_ali、search_ali、print_ali について述べる。

まず関数 register_ali について説明する。該当する箇所を以下に、その定義に用いた変数を表 23 示す。

```
443 void register_ali(ali** ali_root, char* ali_str,const char* original_str) {
444     if(strcmp(ali_str,"alias")==0){
445         printf("alias は引数 1 として指定できません\n");
446         return;
447     }
448   ali* new_ali;
449   ali* selectNode = *ali_root;
450   new_ali = (ali*)malloc(sizeof(ali));
451     if(new_ali == NULL)
452    {
453     printf("ERROR\n");
454     exit(-1);
455    }
456    new_ali->original =(char*)malloc(strlen(original_str)+1);
457    strcpy(new_ali->original, original_str);
458    new_ali->ali_name =(char*)malloc(strlen(ali_str)+1);
459    if(new_ali->ali_name == NULL)
460    {
461     printf("ERROR\n");
462     exit(-1);
463    }
464    strcpy(new_ali->ali_name, ali_str);
465    new_ali->next = NULL;
466
467   if(*ali_root==NULL)
468   {
469    *ali_root = new_ali;
470   }
471
472   else
473   {
474       while (selectNode->next != NULL) selectNode = selectNode->next;
475       selectNode->next = new_ali;
476       new_ali->next =NULL ;
477   }
478   return;
479 }
```

構造体 ali の original に original_str を、ali_name に ali_str を strcpy 関数によりコピーする。そして新たな構造体 ali をリストの末尾に追加するために、追加前にリストの末尾の構造体であった next のアドレスを new_ali に、new_ali->next のアドレスを NULL にする。

表 23: 関数 register_ali の実装で用いた変数一覧

変数名	型	用途
ali_root	ali 型ポインタのポインタ	構造体 ali の先頭のアドレスを示すポインタのポインタ
ali_str	char 型ポインタ	コマンドの別名
original_str	char 型ポインタ	別名設定するコマンドのもともとの名前
new_ali	ali 型ポインタ	新たに追加する構造体
selectNode	ali 型ポインタ	作業用
original_str	char 型ポインタ	別名設定するコマンドのもともとの名前

次に関数 search_ali について説明する。該当する箇所を以下に、その定義に用いた変数を表 24 示す。

```
493 char* search_ali(ali **ali_root, char*name) {
494     ali *selectNode = *ali_root;
495     if(ali_root==NULL)return NULL;
496     char *str = NULL;
497     while(selectNode != NULL) {
498         if(strcmp(selectNode->ali_name,name)==0){
499             str=selectNode->original;
500             return str;
501         }
502         selectNode = selectNode->next;
503     }
504     return "\0";
505 }
```

表 24: 関数 search_ali の実装で用いた変数一覧

変数名	型	用途
ali_root	ali 型ポインタのポインタ	構造体 ali の先頭のアドレスを示すポインタのポインタ
name	char 型ポインタ	別名設定を解除するコマンド名
str	char 型ポインタ	戻り値を格納する
selectNode	ali 型ポインタ	作業用

リストの先頭から name と合致するコマンドの別名を探し続け、合致する別名があればそのコマンドの本来の名前を、合致しなければ終端文字を返す。なお、仕様で言及したように alias コマンドでの別名設定は、複数のコマンドに同一の名称が付けられることは想定していないため、合致した時点で文字列を返し関数を抜ける。

関数 search_ali の戻り値を各コマンド名の文字列比較 strcmp 関数により行なうことで、alias で別名設定されたコマンドであっても本来のコマンドの機能を果たせるようになっている。

そして最後に関数 print_ali について説明する。該当する箇所を以下に、その定義に用いた変数を表 25 示す。

```
481 void print_ali(ali **ali_root) {
482     ali *selectNode = *ali_root;
483     if (ali_root==NULL){
484         printf("alias で設定されているコマンドはありません\n");
485         return;}
486     while(selectNode != NULL) {
487         printf("%s %s\n",selectNode->ali_name,selectNode->original);
488         selectNode = selectNode->next;
489     }
490     return;
491 }
```

表 25: 関数 print_ali の実装で用いた変数一覧

変数名	型	用途
ali_root	ali 型ポインタのポインタ	構造体 ali の先頭のアドレスを示すポインタのポインタ
selectNode	ali 型ポインタ	作業用

ali の先頭アドレスを指す ali_root を selectNode の初期値として用いる。selectNode->ali_name と selectNode->original の中身を出力したあと、selectNode に次の ali のアドレスである selectNode->next を代入し、再度その中身である electNode->ali_name と selectNode->original を示す。この一連の流れを selectNode->next が NULL となるまで続ける。

9.3.2 unalias

unalias コマンドの実装に該当する箇所を以下に示す。

```
795     //unailias
796      else if (strcmp(args[0],command8)==0||
797               strcmp(search_ali(ali_root, args[0]),command8)==0){
798          pre_register_log(log_root,args);
799          if(args[1]==NULL){
800              printf("unalias 引数 1 と書いてください\n");
801              return;
802          }
803          del_ali(ali_root,args[1]);
804          return;
805      }
806     //unailias ここまで
```

unalias コマンドを実行する上で用いた変数を表 26 に示す。

表 26: unalias コマンドの実装で用いた変数一覧

変数名	型	用途
command8	char 型ポインタ	文字列 unalias を示す

またここでは独自に作成した pre_register_log 関数、search_ali 関数が用いられているが、pre_register_log 関数については 5.3.1 章で、search_ali 関数については 9.3.1 章で詳細を説明している。

unalias コマンドは構造体 ali を用いる関数 del_ali 用いて実現した。関数 del_ali について説明する。該当する箇所を以下に、その定義に用いた変数を表 27 に示す。

```
507 void del_ali(ali** ali_root, char* ali_str){
508     ali *selectNode = *ali_root;
509
510     if(ali_root==NULL)return;
511     while(selectNode != NULL) {
512         if(strcmp(selectNode->ali_name,ali_str)==0){
513             selectNode->ali_name=selectNode->original;
514         }
515         selectNode = selectNode->next;
516     }
517     return;
518 }
```

ali の先頭アドレスを指す ali_root を selectNode の初期値として用い、selectNode->next が NULL になるまでループを続ける。unalias の引数と ali の ali_name を strcmp 関数により比較し、合致した場合は ali_name を元のコマンド名に書き換える、すなわち original を代入することにより別名を解除している。したがって、リスト自体を削除しているわけではないため、再度 alias でそのコマンドを別名設定する場合は新たなリストが追加されることになる。

表 27: 関数 del_ali の定義

変数名	型	用途
ali_root	ali 型ポインタのポインタ	構造体 ali の先頭のアドレスを示すポインタのポインタ
selectNode	ali 型ポインタ	作業用
ali_str	char 型のポインタ	unalias の引数

9.4 テスト

9.4.1 テスト方法

仕様や実装方法で説明した箇所が実現されているかを確認する。

9.4.2 テスト結果

実際のテスト結果を以下に示す。

```
~/report2-shiromadaiki$ gcc simple_shell.c
~/report2-shiromadaiki$ ./a.out
Command : alias sl
alias 引数 1 引数 2 と書いてください
Command : alias sl ls
Command : sl -l
total 76
-rwxr-xr-x 1 runner runner 22384 Aug  1 06:47 a.out
-rwxr-xr-x 1 runner runner 22640 Aug  1 02:48 main
-rw-r--r-- 1 runner runner    31 Aug  1 03:20 script_file.txt
-rw-r--r-- 1 runner runner 24081 Aug  1 06:46 simple_shell.c
drwxr-xr-x 1 runner runner    10 Jul 31 01:54 temp
Command : alias kk ss
Command : alias
sl ls
kk ss
Command : unalias
unalias 引数 1 と書いてください
Command : unalias sl
Command : sl
sh: 1: sl: not found
Command : alias
ls ls
kk ss
Command : alias sl ls
Command : alias
ls ls
kk ss
sl ls
```

1 番目の結果より、第二引数が指定されなかった場合エラーメッセージが正しく表示されていることがわかる。

2、3 番目の結果より ls の別名が正しく設定され、オプションを付けても正しく実行されていることがわかる。

4、5 番目の結果より存在しないコマンドについても別名が設定され、引数を指定せずに alias を実行すると別名設定されたコマンドの一覧が正しく表示されていることがわかる。

6 番目の結果より、unalias コマンドで引数が入力されなかった場合のエラーメッセージが正しく表示されていることがわかる。

7、8 番目の結果より unalias により別名設定が解除されていることがわかる。

9、10、11 の結果より unalias により ali 自体が削除されているのではなく、あくまで別名表記が変更されるだけで、再度別名設定をする場合は新たにリストに追加されていることがわかる。

以上の結果より、alias コマンドは正しく実装されていると言える。

10 自分で考えた機能

10.1 機能

今回作成したコマンド名は sum で整数の加算を行い、その結果を表示する。

10.2 仕様

二項以上の加算を行うことができる。また負の数についても計算可能である。扱える数の範囲は最大で 2147483647、最小値で-2147483648 である。

10.3 処理の流れ

引数を算用数字に変換した後に、一つずつ加算する。全ての引数を足し終えた結果を出力する。

10.4 実装方法

sum コマンドの実装に該当する箇所を以下に示す。

```
808     //sum ここから
809      else if(strcmp(args[0],command9)==0||
810              strcmp(search_ali(ali_root, args[0]),command9)==0){
811          pre_register_log(log_root,args);
812          int sum = 0;
813          if(args[1]==NULL||args[2]==NULL){
814              printf("引数は二つ以上指定してください\n");
815              return;
816          }
817          for (int i=1; args[i]!=NULL; i++) {
818             sum += atoi(args[i]);
819           }
820
821          printf("%d\n",sum);
822          return;
823      }
824     //sum ここまで
```

sum コマンドを実装する上で用いた変数を表 28 に示す。

またここでは独自に作成した pre_register_log 関数、search_ali 関数が用いられているが、pre_register_log 関数については 5.3.1 章で、search_ali 関数については 9.3.1 章で詳細を説明している。

args[i] を atoi 関数により int 型の数字に変換し、一つずつ足し合わせそれを sum に格納していく。

10.5 テスト

10.5.1 テスト方法

仕様や実装方法で説明した箇所が実現されているかを確認する。

表 28: sum コマンドの実装で用いた変数一覧

変数名	型	用途
command9	char 型ポインタ	文字列 sum を示す
sum	int 型	各項の和を格納する
i	int 型	args の添字

10.5.2 テスト結果

実際のテスト結果を以下に示す。

```
~/report2-shiromadaiki$ gcc simple_shell.c
~/report2-shiromadaiki$ ./a.out
Command : sum 1 2 4 5 6
18
Command : sum 9.3 4.2
13
Command : sum -1 -2
-3
Command : sum 2147483646 1
2147483647
Command : sum 2147483647 1
-2147483648
Command : sum -2147483648 -1
2147483647
```

1 番目の結果より、全ての引数の加算が行われていることがわかる。

2 番目の結果より、加算は整数のみ対象であることがわかる。

3 番目の結果より、負の整数の加算を行うことも可能であることがわかる。

4、5、6 番目の結果より、扱える数の範囲は最大で 2147483647、最小値で-2147483648 であることがわかる。

以上より、作成したコマンド sum は問題なく機能しているとわかる。

11 工夫点

工夫した点は主に 2 点ある。まず 1 つ目は、エラー処理である。今回実装したコマンドは、引数や実行するタイミングによってエラー処理をしないと正しく動かない場合がある。例えば、コマンドが必要とする引数の数と合致しなかったり、!!コマンドや popd コマンドといった、それらを実行する前に別のコマンドが実行されていることが前提となったりするものである。テスト結果に示している通り、思いつく限りのエラーは想定しているので問題はないはずである。

次に 2 つ目は、オプションの実装である。主に外部コマンドについてではあるが、オプションをつけたコマンドについても実装できるように文字列の扱いに注意を払った。また alias コマンドで別名設定されたコマンドについてもオプションをつけて実行できるようにした。

12 考察

1. 代入とコピーの違い
   あるポインタの要素を別のポインタに渡したい場合、代入ではなく strcmp 関数などを使いコピーする必要がある。というのも、ポインタはアドレスを指すため、代入をしてしまうと欲しい中身だけでなくアドレスまで取得してしまう。その結果、どちらか一方のポインタが指す変数の内容が変化した場合、もう片方のポインタの中身まで変更されてしまう。
   したがって、文字列だけなど要素のみを取得したい場合は代入ではなく、strcpy などを使って要素のみをコピーする必要がある。

2. strcpy 関数とコピー先のサイズ
   strcpy 関数では、コピー先の大きさはコピーの対象となるもののサイズより大きいことが前提となっており、オーバーフローを起こすと関数がエラーを起こしてしまう。したがって、strcpy 関数を実行する前には、malloc 関数などによりコピー先のサイズをコピー元のサイズと同等以上にするなどして確保するか、コピー先の配列の大きさを少し多めに設定するなどの必要がある。
3. ポインタのポインタ今回ポインタのポインタを多く使用したが、これは一つのコマンドを実行するたびに関数 excute_command が終了するからである。関数 excute_command の引数にポインタを用いた場合、ポインタはあるアドレスを指すため、ディレクトリスタックや history、ali などにおいて引数のポインタが参照する地点を変更したとしてもアドレス自体は値渡しなので、関数を抜けるとそれらのポインタは初期化されてしまう。その結果、構造体の先頭を指すはずのポインタが更新されないという事態が起こる。これを解消するのが、ポインタのポインタであった。ポインタのポインタであれば参照渡しとしてアドレスを関数の引数として渡すことができるため、関数内でポインタの参照先が変わって関数を抜けたとしても問題ない。

13 感想

今回の課題はまず全体的に量が多く、コーディングの内容もこれまでの C 言語の講義の内容を包括するようなものであり非常に大変であった。特に難しかった点が三点ある。まず一つ目が、ポインタのエラー処理である。今回はポインタやポインタのポインタなどを多用したため segmentation faults が頻発し、コーディングに苦しんだことである。またこれだけでなく segmentation faults だとエラー箇所がわかりにくくデバッグがとても難しかった。printf を使ったり、ローカル環境の Xcode でデバッガを用いたりするなどしてどうにかデバッグを行った。

次に二つ目は、一つのコマンドの機能が完成したとしても、全体を通して動かしてみると予想だにしないエラーが多発し原因の特定に時間がかかったことである。これまでの課題では、一つのプログラムにつきせいぜい 1〜3 個の機能を持つものを作成することが多かったためあまり実感できなかったが、今回のように多機能を持つプログラムの場合は個々の機能を実装しただ部分的につなぎ合わせていくだけではなく、全体を動かした時の視点を持てるようにしたい。

最後に三つ目は、実行環境による実行結果の違いである。コーディングの効率を考え、プログラム作成の大半はローカル環境の Xcode で行なっていたのだが、全体が完成した後に全く同じコードを replit 上でテストしてみると、得られる結果が異なる、正確には Xcode 上では正しく動いたとしても replit では正しく動かないということが何度かあった。先程の二点と同様に、エラー箇所の特定に非常に時間を要した。特にポインタ関連の記述で問題が多発したが、これは実行環境により許容されている書き方の広さが異なるからであるとわかった。よりよりプログラムを作るには、複数の環境で実行して正しい結果が得られるかを検討するのが良さそうである。

14 謝辞

プログラミング C の授業を担当していただいた、講義担当教員の長谷川 亨教員には、丁寧かつ熱心なご指導を賜りました。今期もオンライン授業でしたが、非常に分かりやすく丁寧な講義を毎週をしてくださりました。ここに感謝の意を表します。

15 作業工程

課題を提出するまでの作業内容を大まかに日付と時間とともに列挙する。

- アルゴリズム設計 (7/13,14)
- プログラム設計 (7/15〜20)
- コーディング (7/21〜24)
- デバッグ (7/25,26)
- レポート作成 (7/27〜7/31)

16　参考文献

参考文献

[1] B.W. カーニハン, D.M. リッチー, 石田晴久訳, ”プログラミング言語 C 第 2 版 ANSI 企画準拠,” 共立出版株式会社, 2017.

[2] C 言語例文集 chdir() カレントディレクトリを変更する, http://cgengo.sakura.ne.jp/chdir.html

[3] C 言語例文集 getcwd() カレントディレクトリを取得する, https://cgengo.sakura.ne.jp/getcwd.html

[4] スタック (stack), http://www.cc.kyoto-su.ac.jp/ yamada/ap/stack.html

[5] 森, 単方向リスト, http://www.ced.is.utsunomiya-u.ac.jp/lecture/2015/prog/p2/kadai2/3_list_1.php

[6] Garfields, ゾンビでもわかる C 言語プログラミング C 言語入門者の応援をします, https://zombie-hunting-club-c.hatenablog.com/entry/2017/11/09/204722,2017-11-09

[7] 長野透,【C 言語入門】文字列内の検索方法まとめ (文字指定、正規表現),SAMURAI ENGINEER,https://www.sejuku.net/blog/25962,2020-06-29

[8] ファイル/指定したディレクトリのファイル一覧を取得する・opendir,readdir, https://c.just4fun.biz/?ファイル/指定したディレクトリのファイル一覧を取得する・opendir%2Creaddir,2015-03-20

[9] ナナ,C 言語 文字列連結【strcat 関数の使い方と 2 つの注意すべきこと, https://monozukuri-c.com/langc-funclist-strcat/#toc6, 2021-02-02

[10] ポインタのポインタ, http://wisdom.sakura.ne.jp/programming/c/c25.html

A　プログラムリスト

作したプログラムを下記に示す。

```
 1 #include <stdio.h>
 2 #include <ctype.h>
 3 #include <stdlib.h>
 4 #include <dirent.h>
 5 #include <sys/types.h>
 6 #include <sys/stat.h>
 7 #include <unistd.h>
 8 #include <sys/wait.h>
 9 #include <string.h>
10 #define BUFLEN    1024     /* コマンド用のバッファの大きさ */
11 #define MAXARGNUM  256     /* 最大の引数の数 */
12 #define PATHNAME_SIZE 512
13 #define MAXLENGTH 256
14
15 typedef struct Node{
16   char *data;
17   struct Node *next;
18 }stack;
19 void push(stack**, char*);
20 void print_stack(stack** );
21 char* pop(stack **);
22
23 typedef struct command_log{
24   char *data;
25   struct command_log *next;
26 }hisotry;
27 void pre_register_log(hisotry**,char**);
28 void register_log(hisotry**,char*);
29 void print_log(hisotry**);
30 char* latest_log(hisotry **);
31 char* search_log(hisotry **,char*);
```

```
32
33 typedef struct alias{
34   char *original;
35   char *ali_name;
36   struct alias *next;
37 }ali;
38 void register_ali(ali**,char*,const char*);
39 void print_ali(ali**);
40 char* search_ali(ali**,char*);
41 void del_ali(ali** ,char* );
42
43 int parse(char [], char *[]);
44 void execute_command(char *[],int,stack **,hisotry**,ali **,char**);
45 int main(int argc, char *argv[])
46 {
47     char command_buffer[BUFLEN]; /* コマンド用のバッファ */
48     char *args[MAXARGNUM];       /* 引数へのポインタの配列 */
49     int command_status;          /* コマンドの状態を表す
50                                     command_status = 0 : フォアグラウンドで実行
51                                     command_status = 1 : バックグラウンドで実行
52                                     command_status = 2 : シェルの終了
53                                     command_status = 3 : 何もしない */
54     int cc;
55     FILE *fp;
56     char string[MAXLENGTH], *tmp;
57     stack* stack_root = NULL;
58     hisotry*  log_root=NULL;
59     ali* ali_root=NULL;
60     char init_name[]="Command";
61     char *prompt_name=init_name;
62     /*
63      *  無限にループする
64      */
65     //スクリプト機能
66
67       /*  fp=fopen("script_file.txt","r");
68         if(fp == NULL){
69                 fprintf(stderr, "script_file.txt is not found.\n");
70             }
71         else{
72         cc = getc(fp);                              // ファイルからの 1 文字読みだし
73
74            while(cc != EOF){
75
76                while(isspace(cc))  cc = getc(fp);
77
78                if(cc == EOF) break;
79
80                tmp = string;
81                while(!(isspace(cc) || cc == EOF)){
82                   if((tmp - string) >= (MAXLENGTH - 1)){
83                        fprintf(stderr, "Too long sentence.\n");
84                        fclose(fp);
85                        exit(1);
86                    }
87                    *tmp++ = cc;
88                    cc = getc(fp);
89                }
90                *tmp = '\0';
91                system(string);
92                }
93         fclose(fp);
```

```
 94              return 0;
 95          }*/
 96      //スクリプト機能ここまで
 97
 98
 99
100      for(;;) {
101
102
103          /*
104           *  プロンプトを表示する
105           */
106
107          printf("%s : ",init_name);
108
109          /*
110           *  標準入力から１行を command_buffer へ読み込む
111           *  入力が何もなければ改行を出力してプロンプト表示へ戻る
112           */
113
114          if(fgets(command_buffer, BUFLEN, stdin) == NULL) {
115              printf("\n");
116              continue;
117          }
118
119          /*
120           *  入力されたバッファ内のコマンドを解析する
121           *
122           *  返り値はコマンドの状態
123           */
124
125          command_status = parse(command_buffer, args);
126
127          /*
128           *  終了コマンドならばプログラムを終了
129           *  引数が何もなければプロンプト表示へ戻る
130           */
131
132          if(command_status == 2) {
133              printf("done.\n");
134              exit(EXIT_SUCCESS);
135          } else if(command_status == 3) {
136              continue;
137          }
138
139          /*
140           *  コマンドを実行する
141           */
142
143          execute_command(args,command_status,&stack_root
144                          ,&log_root,&ali_root,&prompt_name);
145      }
146      free(tmp);
147      free(stack_root);
148      free(log_root);
149      free(ali_root);
150      return 0;
151  }
152
153  /*---------------------------------------------------------------------------
154   *
155   *  関数名    : parse
```

```
156  *
157  *  作業内容 : バッファ内のコマンドと引数を解析する
158  *
159  *  引数     :
160  *
161  *  返り値   : コマンドの状態を表す :
162  *                0 : フォアグラウンドで実行
163  *                1 : バックグラウンドで実行
164  *                2 : シェルの終了
165  *                3 : 何もしない
166  *
167  *  注意     :
168  *
169  *-----------------------------------------------------------------------------*/
170
171 int parse(char buffer[],        /* バッファ */
172           char *args[])         /* 引数へのポインタ配列 */
173 {
174     int arg_index;   /* 引数用のインデックス */
175     int status;   /* コマンドの状態を表す */
176
177     /*
178      *  変数の初期化
179      */
180
181     arg_index = 0;
182     status = 0;
183
184     /*
185      *  バッファ内の最後にある改行をヌル文字へ変更
186      */
187
188     *(buffer + (strlen(buffer) - 1)) = '\0';
189
190     /*
191      *  バッファが終了を表すコマンド（"exit"）ならば
192      *  コマンドの状態を表す返り値を 2 に設定してリターンする
193      */
194
195     if(strcmp(buffer, "exit") == 0) {
196
197         status = 2;
198         return status;
199     }
200
201     /*
202      *  バッファ内の文字がなくなるまで繰り返す
203      *  （ヌル文字が出てくるまで繰り返す）
204      */
205
206     while(*buffer != '\0') {
207
208         /*
209          *  空白類（空白とタブ）をヌル文字に置き換える
210          *  これによってバッファ内の各引数が分割される
211          */
212
213         while(*buffer == ' ' || *buffer == '\t') {
214             *(buffer++) = '\0';
215         }
216
217         /*
```

```
218          * 空白の後が終端文字であればループを抜ける
219          */
220
221         if(*buffer == '\0') {
222             break;
223         }
224
225         /*
226          *  空白部分は読み飛ばされたはず
227          *  buffer は現在は arg_index + 1 個めの引数の先頭を指している
228          *
229          *  引数の先頭へのポインタを引数へのポインタ配列に格納する
230          */
231
232         args[arg_index] = buffer;
233         ++arg_index;
234
235         /*
236          *  引数部分を読み飛ばす
237          *  （ヌル文字でも空白類でもない場合に読み進める）
238          */
239
240         while((*buffer != '\0') && (*buffer != ' ') && (*buffer != '\t')) {
241             ++buffer;
242         }
243     }
244
245     /*
246      *  最後の引数の次にはヌルへのポインタを格納する
247      */
248
249     args[arg_index] = NULL;
250
251     /*
252      *  最後の引数をチェックして "&" ならば
253      *
254      *  "&" を引数から削る
255      *  コマンドの状態を表す status に 1 を設定する
256      *
257      *  そうでなければ status に 0 を設定する
258      */
259
260     if(arg_index > 0 && strcmp(args[arg_index - 1], "&") == 0) {
261
262         --arg_index;
263         args[arg_index] = NULL;
264         status = 1;
265
266     } else {
267
268         status = 0;
269
270     }
271
272     /*
273      *  引数が何もなかった場合
274      */
275
276     if(arg_index == 0) {
277         status = 3;
278     }
279
```

```
280     /*
281      *  コマンドの状態を返す
282      */
283
284     return status;
285 }
286
287 void push(stack** stack_root, char* str) {
288   stack* new_stack;
289   new_stack = (stack*)malloc(sizeof(stack));
290     if(new_stack == NULL)
291    {
292     printf("ERROR\n");
293     exit(-1);
294    }
295    new_stack->data =(char*)malloc(strlen(str)+1);
296    if(new_stack->data == NULL)
297    {
298     printf("ERROR\n");
299     exit(-1);
300    }
301    strcpy(new_stack->data, str);
302    new_stack->next = NULL;
303   if(*stack_root==NULL)
304   {
305    *stack_root = new_stack;
306   }
307   else
308   {
309       new_stack->next =*stack_root ;
310       *stack_root = new_stack;
311   }
312   return;
313 }
314
315 char* pop(stack **stack_root){
316   char *directory;
317   stack *next, *fr;
318     next = (stack*)malloc(sizeof(stack));
319   if(next == NULL){
320     printf("ERROR\n");
321     exit(-1);
322   }
323   if(*stack_root == NULL){
324     return NULL;
325   }
326   directory = (*stack_root) -> data;
327   next = (*stack_root) -> next;
328   fr = *stack_root;
329   *stack_root = next;
330   free(fr);
331   return directory;
332 }
333
334 void print_stack(stack **stack_root) {
335     stack *selectNode = *stack_root;
336     printf("Directory stack = ");
337     while(selectNode != NULL) {
338         printf("%s",selectNode->data);
339         selectNode = selectNode->next;
340         if( selectNode != NULL ) {
341             printf(", ");
```

```
342         }
343     }
344     printf("\n");
345     return;
346 }
347
348 void pre_register_log(hisotry** log_root,char* args[]){
349     int i=1;
350     char command[100];
351     command[0]='\0';
352     strcpy(command, args[0]);
353     while (args[i]!=NULL) {
354           strcat(command," ");
355           strcat(command, args[i]);
356         if(args[i+1]==NULL)break;
357         i++;
358       }
359       register_log(log_root,command);
360 }
361
362 void register_log(hisotry** log_root, char* str) {
363   hisotry* new_history;
364   hisotry* selectNode = *log_root;
365   new_history = (hisotry*)malloc(sizeof(hisotry));
366     if(new_history == NULL)
367    {
368     printf("ERROR\n");
369     exit(-1);
370    }
371    new_history->data =(char*)malloc(strlen(str)+1);
372    if(new_history->data == NULL)
373    {
374     printf("ERROR\n");
375     exit(-1);
376    }
377    strcpy(new_history->data, str);
378    new_history->next = NULL;
379
380   if(*log_root==NULL)
381   {
382    *log_root = new_history;
383   }
384
385   else
386   {
387       int count=1;
388       while (selectNode->next != NULL) {
389             count++;
390             selectNode = selectNode->next;
391         }
392         if(count==32){
393           printf("history の数が上限です\n");
394         }
395         else{
396       selectNode->next = new_history;
397       new_history->next =NULL ;
398         }
399   }
400   return;
401 }
402
403 void print_log(hisotry **log_root) {
```

```
404     hisotry *selectNode = *log_root;
405     int i=1;
406     while(selectNode != NULL) {
407         printf("%d %s\n",i,selectNode->data);
408         selectNode = selectNode->next;
409         i++;
410     }
411     return;
412 }
413
414 char* latest_log(hisotry **log_root) {
415     hisotry *selectNode = *log_root;
416     if(selectNode==NULL){
417         printf("historyが空です\n");
418         return NULL;
419     }
420     hisotry *p = NULL;
421     while(selectNode != NULL) {
422         p=selectNode;
423         selectNode = selectNode->next;
424     }
425     return p->data;
426 }
427 char* search_log(hisotry **log_root,char*name) {
428     hisotry *selectNode = *log_root;
429     hisotry *p = NULL;
430     char *str = NULL;
431     while(selectNode != NULL) {
432         p=selectNode;
433         if(p->data[0]==*(name+0)){
434             if(strstr(p->data,name)!=NULL){
435             str=strstr(p->data,name);
436             }}
437         selectNode = selectNode->next;
438         if (p==NULL)return NULL;
439     }
440     return str;
441 }
442
443 void register_ali(ali** ali_root, char* ali_str,const char* original_str) {
444     if(strcmp(ali_str,"alias")==0){
445         printf("aliasは引数1として指定できません\n");
446         return;
447     }
448   ali* new_ali;
449   ali* selectNode = *ali_root;
450   new_ali = (ali*)malloc(sizeof(ali));
451     if(new_ali == NULL)
452    {
453     printf("ERROR\n");
454     exit(-1);
455    }
456    new_ali->original =(char*)malloc(strlen(original_str)+1);
457    strcpy(new_ali->original, original_str);
458    new_ali->ali_name =(char*)malloc(strlen(ali_str)+1);
459    if(new_ali->ali_name == NULL)
460    {
461     printf("ERROR\n");
462     exit(-1);
463    }
464    strcpy(new_ali->ali_name, ali_str);
465    new_ali->next = NULL;
```

```
466
467   if(*ali_root==NULL)
468   {
469    *ali_root = new_ali;
470   }
471
472   else
473   {
474       while (selectNode->next != NULL) selectNode = selectNode->next;
475       selectNode->next = new_ali;
476       new_ali->next =NULL ;
477   }
478   return;
479 }
480
481 void print_ali(ali **ali_root) {
482     ali *selectNode = *ali_root;
483     if (ali_root==NULL){
484         printf("alias で設定されているコマンドはありません\n");
485         return;}
486     while(selectNode != NULL) {
487         printf("%s %s\n",selectNode->ali_name,selectNode->original);
488         selectNode = selectNode->next;
489     }
490     return;
491 }
492
493 char* search_ali(ali **ali_root, char*name) {
494     ali *selectNode = *ali_root;
495     if(ali_root==NULL)return NULL;
496     char *str = NULL;
497     while(selectNode != NULL) {
498         if(strcmp(selectNode->ali_name,name)==0){
499             str=selectNode->original;
500             return str;
501         }
502         selectNode = selectNode->next;
503     }
504     return "\0";
505 }
506
507 void del_ali(ali** ali_root, char* ali_str){
508     ali *selectNode = *ali_root;
509
510     if(ali_root==NULL)return;
511     while(selectNode != NULL) {
512         if(strcmp(selectNode->ali_name,ali_str)==0){
513             selectNode->ali_name=selectNode->original;
514         }
515         selectNode = selectNode->next;
516     }
517     return;
518 }
519 /*-------------------------------------------------------------------------
520  *
521  *  関数名    : execute_command
522  *
523  *  作業内容 : 引数として与えられたコマンドを実行する
524  *               コマンドの状態がフォアグラウンドならば、コマンドを
525  *               実行している子プロセスの終了を待つ
526  *               バックグラウンドならば子プロセスの終了を待たずに
527  *               main 関数に返る（プロンプト表示に戻る）
```

```
528  *
529  *  引数     :
530  *
531  *  返り値   :
532  *
533  *  注意     :
534  *
535  *-------------------------------------------------------------------------*/
536 void execute_command(char *args[],    /* 引数の配列 */
537                      int command_status,stack **stack_root,
538                      hisotry** log_root,ali** ali_root, char**prompt_name)     /* コ
マンドの状態 */
539 {
540     char *pcommand1="cd";
541     char command1[30];
542     command1[0]='\0';
543     strcpy(command1,pcommand1);
544
545     char *pcommand2="pushd";
546     char command2[30];
547     command2[0]='\0';
548     strcpy(command2,pcommand2);
549
550     char *pcommand3="dirs";
551     char command3[30];
552     command3[0]='\0';
553     strcpy(command3,pcommand3);
554
555     char *pcommand4="popd";
556     char command4[30];
557     command4[0]='\0';
558     strcpy(command4,pcommand4);
559
560     char *pcommand5="history";
561     char command5[30];
562     command5[0]='\0';
563     strcpy(command5,pcommand5);
564
565     char *pcommand6="prompt";
566     char command6[30];
567     command6[0]='\0';
568     strcpy(command6,pcommand6);
569
570     char *pcommand7="alias";
571     char command7[30];
572     command7[0]='\0';
573     strcpy(command7,pcommand7);
574
575     char *pcommand8="unalias";
576     char command8[30];
577     command8[0]='\0';
578     strcpy(command8,pcommand8);
579
580     char *pcommand9="sum";
581     char command9[30];
582     command9[0]='\0';
583     strcpy(command9,pcommand9);
584
585
586     char pathname[PATHNAME_SIZE];  // ファイルパス
587     memset(pathname, '\0', PATHNAME_SIZE);
588
```

```
589
590 hisotry:
591     //cd ここから
592     if(strcmp(args[0],command1)==0||
593        strcmp(search_ali(ali_root, args[0]),command1)==0)
594     {
595         pre_register_log(log_root,args);
596     // カレントディレクトリ取得
597     getcwd(pathname, PATHNAME_SIZE);
598     fprintf(stdout,"変更前のファイルパス:%s\n", pathname);
599
600       if (args[1]==NULL)
601        {
602
603     // カレントディレクトリ変更
604        char* str = getenv( "HOME");
605        if( str == NULL ){
606         fputs( "環境変数の取得に失敗しました。\n", stderr );
607         exit(1);
608        }
609        chdir(str); // チェンジディレクトリ
610        getcwd(str, PATHNAME_SIZE);
611        fprintf(stdout,"現在のファイルパス:%s\n", str);
612            return;
613       }
614       else
615       {
616         // カレントディレクトリ変更
617        if(chdir(args[1])==-1)
618        {
619         printf("No such file or directory\n");
620        }
621        else
622        {
623         getcwd(pathname, PATHNAME_SIZE);
624         fprintf(stdout,"現在のファイルパス:%s\n", pathname);
625        }
626       }
627         strcat(command1," ");
628         strcat(command1,args[1]);
629         return;;
630     }
631       // cd ここまで
632
633       //pushd
634     else if(strcmp(args[0],command2)==0||
635             strcmp(search_ali(ali_root, args[0]),command2)==0)
636     {
637         pre_register_log(log_root,args);
638         getcwd(pathname, PATHNAME_SIZE);
639         char *str=pathname;
640         push(stack_root,str);
641     }
642       //pushd ここまで
643
644       //dirs
645      else if(strcmp(args[0],command3)==0||
646              strcmp(search_ali(ali_root, args[0]),command3)==0)
647      {
648         pre_register_log(log_root,args);
649         print_stack(stack_root);
650      }
```

```
651      //dirs ここまで
652
653      //popd
654      else if(strcmp(args[0],command4)==0||
655              strcmp(search_ali(ali_root, args[0]),command4)==0)
656      {
657       pre_register_log(log_root,args);
658       char *str;
659       str=pop(stack_root);
660          if(str==NULL){
661              printf("ディレクトリスタックが空です\n");
662              return;
663          }
664       chdir(str);
665       getcwd(str, PATHNAME_SIZE);
666       fprintf(stdout,"現在のファイルパス:%s\n", str);
667      }
668      //popd ここまで
669
670      //histotry
671      else if(strcmp(args[0],command5)==0||
672              strcmp(search_ali(ali_root, args[0]),command5)==0){
673         print_log(log_root);
674         register_log(log_root,command5);
675      }
676     // !系コマンド
677
678      else if (args[0][0]=='!'){
679        if (args[0][1]=='!'){
680            if( latest_log(log_root)==NULL) return;
681         strcpy(args[0],latest_log(log_root));
682         int j=0;
683          char *p1,*p2;
684          p1 =(char*)malloc(strlen(args[0])+1);
685          if(p1 == NULL)
686           {
687               printf("ERROR\n");
688               exit(-1);
689           }
690          strcpy(p1,args[0]);
691          p2 = strtok(p1, " ");
692          while (p2)
693                  {
694                      if(j!=0){
695                          args[j] =(char*)malloc(strlen(args[0])+1);
696                          if(args[j] == NULL)
697                               {
698                               printf("ERROR\n");
699                               exit(-1);
700                               }
701                      }
702                          strcpy(args[j],p2);
703                          p2 = strtok(NULL, " ");
704                      j++;
705                  }
706         goto hisotry;
707         }
708
709        if (*(args[0]+1)=='\0'){
710          printf("!の後に文字を続けてください\n");
711          return;
712         }
```

```
713         int i=1;
714          char tmp[10];
715         while(1) {
716             if(i==1){
717                 strcpy(tmp,&args[0][i]);
718                 i++;}
719             else{
720             strcat(tmp,&args[0][i]);
721              i++;
722             if (*(args[0]+i)=='\0') break;
723           }
724         }
725
726          if( search_log(log_root,tmp)==NULL) {
727              printf("該当するコマンドがありません\n");
728              return;}
729            strcpy(args[0],search_log(log_root,tmp));
730          //複数行ある場合 args[0] に全て入ってしまっている
731          int j=0;
732          char *p1,*p2;
733          p1 =(char*)malloc(strlen(args[0])+1);
734          if(p1 == NULL)
735           {
736               printf("ERROR\n");
737               exit(-1);
738           }
739          strcpy(p1,args[0]);
740          p2 = strtok(p1, " ");
741          while (p2)
742                  {
743                      if(j!=0){
744                          args[j] =(char*)malloc(strlen(args[0])+1);
745                          if(args[j] == NULL)
746                               {
747                               printf("ERROR\n");
748                               exit(-1);
749                               }
750                      }
751                          strcpy(args[j],p2);
752                          p2 = strtok(NULL, " ");
753                      j++;
754                  }
755            goto hisotry;
756      }
757     //!系コマンドここまで
758
759     //prompt
760      else if (strcmp(args[0],command6)==0||
761               trcmp(search_ali(ali_root, args[0]),command6)==0){
762          pre_register_log(log_root,args);
763          if (args[1]!=NULL){
764              char *temp=args[1];
765              strcpy(*prompt_name,temp);
766              return;
767          }
768          else{
769              char original_name[]="Command";
770              char *init_name=original_name;
771              strcpy(*prompt_name,init_name);
772              return;
773          }
774         //prompt ここまで
```

```
775      }
776      //alias
777      else if (strcmp(args[0],command7)==0||
778               strcmp(search_ali(ali_root, args[0]),command7)==0){
779          pre_register_log(log_root,args);
780          if(args[1]==NULL){
781              print_ali(ali_root);
782              return;
783          }
784          else {
785              if(args[2]==NULL) {
786                  printf("alias 引数 1 引数 2 と書いてください\n");
787                  return;
788              }
789              register_ali(ali_root,args[1],args[2]);
790              return;
791          }
792      }
793     //alias ここまで
794
795     //unailias
796      else if (strcmp(args[0],command8)==0||
797               strcmp(search_ali(ali_root, args[0]),command8)==0){
798          pre_register_log(log_root,args);
799          if(args[1]==NULL){
800              printf("unalias 引数 1 と書いてください\n");
801              return;
802          }
803          del_ali(ali_root,args[1]);
804          return;
805      }
806     //unailias ここまで
807
808     //sum ここから
809      else if(strcmp(args[0],command9)==0||
810              strcmp(search_ali(ali_root, args[0]),command9)==0){
811          pre_register_log(log_root,args);
812          int sum = 0;
813          if(args[1]==NULL||args[2]==NULL){
814              printf("引数は二つ以上指定してください\n");
815              return;
816          }
817          for (int i=1; args[i]!=NULL; i++) {
818             sum += atoi(args[i]);
819           }
820
821          printf("%d\n",sum);
822          return;
823      }
824     //sum ここまで
825
826       //自作コマンド以外
827       else
828       {
829         pre_register_log(log_root,args);
830         int i=1;
831         char command[100];
832         command[0]='\0';
833         strcpy(command, args[0]);
834         char *ali_command,*temp;
835         ali_command=(char*)malloc(strlen(search_ali(ali_root, args[0]))+1);
836         if(ali_command == NULL)
```

```
837         {
838           printf("ERROR\n");
839           exit(-1);
840         }
841         temp=(char*)malloc(strlen(command)+1);
842          if(temp == NULL)
843         {
844           printf("ERROR\n");
845           exit(-1);
846         }
847         strcpy(ali_command,search_ali(ali_root, args[0]));
848
849           if( strcmp(ali_command,"\0")!=0) strcpy(command, ali_command);
850
851           while (args[i]!=NULL) {
852               strcat(command," ");
853               //ここに*の処理
854               if(strcmp(args[i],"*")==0){
855                   struct stat    filestat;
856                   struct dirent *directory;
857                   DIR           *dp;
858                      dp = opendir(".");
859                         while((directory=readdir(dp))!=NULL){
860                           if(!strcmp(directory->d_name, ".") ||
861                              !strcmp(directory->d_name, ".."))
862                             continue;
863                           if(stat(directory->d_name,&filestat)){
864                             perror("main");
865                             exit(1);
866                           }else{
867                               strcpy(args[i],directory->d_name);
868                               strcpy(temp,command);
869                               strcat(command, args[i]);
870                               system(command);
871                               strcpy(command,temp);
872                           }
873                         }
874                         closedir(dp);
875                   return;
876               }
877               strcat(command, args[i]);
878             if(args[i+1]==NULL)break;
879             i++;
880           }
881           system(command);
882           return;
883     }
884     return;
885 }
886
887 /*-- END OF FILE -------------------------------------------------------------*/
```