プログラミングC演習報告書
プログラミングC 第2回レポート課題

【担当教員】　天野 辰哉・長谷川　亨・小泉 佑揮　教員

【提出者】　安西俊輔　(09B22002)
計算機コース・2 年
u262691h@ecs.osaka-u.ac.jp
【提出日】　2023 年 8 月 1 日

目 次

1　課題内容

課題: サブセット版のシェルを作成する

課題内容のまとめ：

サブセット版のシェルを C 言語を用いて作成する。このサブセット版のシェルには以下の機能が含まれる。

1. 外部コマンドの実行機能：指定した別のプログラムを実行できるようにする。

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できるようにする。- 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できるようにする。- dirs コマンド：現在のディレクトリスタックを表示できるようにする。- popd コマンド：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し、スタックから削除する。

3. ヒストリ機能：- history コマンド：実行したコマンドを実行した順番とともに表示できるようにする。- !!コマンド：1 つ前のコマンドを再度実行できるようにする。- !string コマンド：指定した文字列で始まる最新のコマンドを再度実行できるようにする。

4. ワイルドカード機能：”*” によってファイル名を補完する機能を実現する。

5. プロンプトの変更機能：prompt コマンドによってプロンプトを変更できるようにする。

6. スクリプト機能：テキストファイルからコマンドを読み込み、実行できるようにする。

7. エイリアス機能：alias コマンドによってコマンドの別名を設定し、unalias コマンドで解除できるようにする。

自分で考えた機能（1 つ）：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　プログラム全体の全体説明

2.1　仕様

簡易版シェルの主な仕様は以下の通りです。

1. プロンプトの表示：プログラムはプロンプトを表示して、ユーザにコマンドの入力を促す。
2. コマンドの読み込み：ユーザがコマンドを入力すると、その入力行が文字列としてバッファに格納される。
3. コマンドの解析：入力されたコマンド文字列は、parse 関数によって解析され、コマンドと引数の配列に分割される。
4. コマンドの判別：プログラムは解析されたコマンドを判別し、対応する処理を行う。
5. 外部コマンドの実行：入力が外部コマンド（例：ls、echo、cat など）の場合、execvp 関数を使用して指定された外部プログラムを実行する。
6. ディレクトリの管理：入力がディレクトリ関連のコマンド（cd、pushd、popd、dirs）の場合、対応する関数（change_directory、push_directory、pop_directory、print_directory_stack）を呼び出してディレクトリを操作する。

7. ヒストリ機能：入力がヒストリ関連のコマンド（history、!!、!文字列）の場合、対応する関数（history、exclamation、string）を呼び出して過去のコマンドを処理する。
8. プロンプトの変更：入力がプロンプト関連のコマンド（prompt）の場合、対応する関数（prompt）を呼び出してプロンプト文字列を変更する。
9. エイリアス機能：入力がエイリアス関連のコマンド（alias、unalias）の場合、対応する関数（alias、unalias）を呼び出してエイリアスを設定または解除する。
10. ワイルドカード機能：入力がワイルドカードを含む場合、対応する関数（wild）を呼び出してファイル名にマッチするコマンドを実行する。
11. ブラックジャックゲーム：入力がブラックジャック関連のコマンド（bj）の場合、対応する関数（blackjack）を呼び出してブラックジャックゲームをプレイする。
12. スクリプト機能：この機能のみ main 関数のみで処理を行う。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。
13. 無限ループの再開：処理が終了すると、プログラムは再びプロンプトの表示に戻り、次のコマンドの入力を待つ。

2.2 処理の流れ (処理概要)

簡潔に処理の流れを以下に示す。

1. プロンプトを表示し、コマンドが文字列として格納される。
2. 文字列をヒストリに保存する。
3. 入力された文字列を引数とコマンドに分割する。
4. コマンドを判定しそれぞれの関数で実行する。
5. 正常に実行されれば、以上の作業をループをする。

2.3 実装方法

1. プロンプトの文字列 prompt_string を表示し、コマンドの文字列 command_buffer に格納する。
2. ヒストリの二次元配列 past_command に格納する。
3. parse 関数でコマンドと引数に分け args に格納する。
4. if 文で判定し、それぞれの関数を実行する。
5. for 文でループする。

3　外部コマンド実行機能

3.1　仕様

・ユーザが入力したコマンドは、シェルによって解析され、外部コマンドとその引数の配列に分割される。
・シェルは外部コマンドを実行し、その実行結果を表示する。
・ユーザがコマンドの末尾に ”&” を付けると、外部コマンドをバックグラウンドで実行する。
・バックグラウンド実行では、外部コマンドの終了を待たずに次のコマンドの入力待ち状態に戻る。

3.2　処理の流れ (処理概要)

1. ユーザがコマンドを入力すると、main 関数はそのコマンド文字列を受け取る。
2. main 関数は入力文字列を parse 関数に渡し、外部コマンドと引数の配列に分割する。
3. parse 関数は、コマンド文字列を解析してコマンドの状態を返す（フォアグラウンド or バックグラウンド or シェルの終了）。
4. main 関数は parse 関数から受け取ったコマンドの状態に応じて、
5. execute_command 関数を呼び出す。
6. execute_command 関数は、外部コマンドを fork して子プロセスで実行する。バックグラウンド実行の場合、親プロセス（シェル）は子プロセスの終了を待たずに次のコマンドの入力待ち状態に戻る。

3.3　実装方法

外部コマンド実行は、fork と execvp を組み合わせて行う。
fork 関数は新しいプロセスを生成し、親プロセスと子プロセスの両方で実行を続ける。
子プロセスは execvp 関数を用いて、外部コマンドを実行する。execvp は PATH 変数を参照して外部コマンドを検索・実行する。
親プロセスは waitpid 関数を使用して子プロセスの終了を待つ（フォアグラウンド実行の場合のみ）。
バックグラウンド実行では、子プロセスの終了を待たずにプロンプトに戻る。

3.4　テスト

以下は mkdir と ls の実行結果である。

Command : ls
a.out Makefile replit.nix
A.txt mysh script.txt
B.txt mysh.c
main.c perfect.c
Command : mkdir A_dir
Command : ls
$A_dirmain.cperfect.c$
a.out Makefile replit.nix
A.txt mysh script.txt
B.txt mysh.c

以上より、外部コマンド機能は正常に動作しているといえる。

4 ディレクトリの管理機能

4.1 仕様

・cd コマンドを使用して、カレントディレクトリを変更できる。引数がない場合は親ディレクトリに移動する。
・pushd コマンドを使用して、カレントディレクトリをディレクトリスタックにプッシュする。
・dirs コマンドを使用して、ディレクトリスタックの内容を表示する。
・popd コマンドを使用して、ディレクトリスタックから最後にプッシュされたディレクトリをポップしてカレントディレクトリに移動する。

4.2 処理の流れ (処理概要)

1. ユーザが cd コマンドを入力すると、change_directory 関数が呼び出される。
2. change_directory 関数は、引数によってカレントディレクトリを変更する。引数がない場合は chdir 関数を使って親ディレクトリに移動する。
3. ユーザが pushd コマンドを入力すると、push_directory 関数が呼び出される。
4. push_directory 関数は、現在のカレントディレクトリを取得し、ディレクトリスタックにプッシュする。
5. ユーザが dirs コマンドを入力すると、print_directory_stack 関数が呼び出される。
6. print_directory_stack 関数は、ディレクトリスタックの内容を表示する。ユーザが popd コマンドを入力すると、pop_directory 関数が呼び出される。
7. pop_directory 関数は、ディレクトリスタックから最後にプッシュされたディレクトリをポップしてカレントディレクトリに移動する。

4.3 実装方法

chdir 関数を使用して、change_directory 関数でカレントディレクトリを変更する。
カレントディレクトリの保存と復元には、getcwd 関数と chdir 関数を組み合わせる。
push_directory 関数では、配列 directory_stack にカレントディレクトリを追加し、stack_index を 1 増やしている。
print_directory_stack 関数では、配列 directory_stack の内容を順番に表示する。
pop_directory 関数では、chdir 関数を用いてカレントディレクトリを配列 directory_stack の最も後ろに格納された場所に飛ぶ。

4.4 テスト

以下は cd(引数なし) の実行結果である。

```
Command : pwd
/home/runner/report2-u262691h
Command : cd
Command : pwd
/home/runner
```

以下は cd(引数あり) の実行結果である。

```
Command : pwd
/home/runner/report2-u262691h
Command : cd A_dir
Command : pwd
/home/runner/report2-u262691h/A_dir
```

以下は pushd,dirs,popd の実行結果である。

```
Command : pwd
/home/runner/report2-u262691h
Command : pushd
Command : dirs
/home/runner/report2-u262691h
Command : cd A_dir
Command : pwd
/home/runner/report2-u262691h/A_dir
Command : popd
Command : pwd
/home/runner/report2-u262691h
```

以上よりディレクトリ機能は正常に動作している。

5　ヒストリー機能

5.1　仕様

- history コマンドを使用して、過去に実行されたコマンドの履歴を表示する。
- !! コマンドを使用して、直前に実行されたコマンドを再実行する。
- !string コマンドを使用して、指定した文字列で始まる最近のコマンドを再実行する。コマンドの履歴は、配列 past_command に保存される。

5.2　処理の流れ (処理概要)

1. ユーザが history コマンドを入力すると、history 関数が呼び出される。
2. history 関数は、配列 past_command 内の過去に実行されたコマンドを順番に表示する。
3. ユーザが !! コマンドを入力すると、exclamation 関数が呼び出される。
4. exclamation 関数は、配列 past_command の最後に保存されたコマンドを取得し、そのコマンドを再実行する。
5. ユーザが !string コマンドを入力すると、string 関数が呼び出される。
6. string 関数は、指定した文字列で始まる最後のコマンドを配列 past_command から検索し、見つかったコマンドを再実行する。

5.3　実装方法

コマンドの履歴は、配列 past_command に保存され、変数 past_index で履歴の数を管理する。
まず、main 関数で command_buffer を配列 past_command にコピーする。このとき、格納しているコマンド数が 32 以上であれば配列 past_command に格納されているコマンドをひとつずつ前に上書きコピーし、そして command_vuffer を 32 個目にコピーすると配列が最近の 32 個のコマンドに更新できた。コピーを終えると past_index を 1 増やす。
history 関数では、past_command 配列内のコマンドをループして順番に表示する。
exclamation 関数では、past_command 配列の最後に保存されたコマンドを再実行する。再実行には再度パースしてコマンドを実行する。
!string の場合は、文字列を args[1] に格納するため、command_buffer の string の場所をひとつ後ろにずらして”!”と string の間に空白をいれてから parse 関数に渡す。string 関数では、指定した文字列の長さを k として k 文字分 string と一致するコマンドを past_command 配列内から strncmp 検索し、見つかったコマンドを再実行する。見つからなかった場合はエラーメッセージを表示する。

5.4 テスト

以下は history の実行結果である。

```
Command : ls
A_dir main.c perfect.c
a.out Makefile replit.nix
A.txt mysh script.txt
B.txt mysh.c
Command : pwd
/home/runner/report2-u262691h
Command : history
1: ls
2: pwd
3: history
```

以下は!!と history の実行結果である。!!が正常に動作している場合に、再度実行したコマンドが history に格納されていることを示すために history を実行した。

```
Command : pwd
/home/runner/report2-u262691h
Command : !!
/home/runner/report2-u262691h
Command : history
1: pwd
2: pwd
3: history
```

以下は!string の実行結果である。今回は!p を入力し、２個前の pwd コマンドが再度実行されることを示した。

```
Command : pwd
/home/runner/report2-u262691h
Command : cd
Command : ls
report2-u262691h
Command : !p
/home/runner
```

以上よりヒストリー機能が正常に動作していることがわかる。

6　ワイルドカード機能

6.1　仕様

- ユーザが”*”を含むコマンドを入力すると、ワイルドカードを展開してマッチするファイルを一括して処理する。

6.2　処理の流れ (処理概要)

1. ユーザがコマンドを入力すると、パースが入力されたコマンドを解析し、ワイルドカードが含まれているかをチェックする。ワイルドカードが含まれている場合、ワイルドカードを展開してファイル名を取得する。
2. 取得したファイル名を元に、コマンドの引数を置き換える。
3. 引数が置き換えられたのちコマンドを実行する。

6.3　実装方法

ワイルドカードの展開は、wild 関数が担当します。
wild 関数では、opendir 関数を使用してカレントディレクトリを開き、readdir 関数を使用してファイル名を取得します。
取得したファイル名の中からワイルドカードにマッチするファイル名を検索します。
ワイルドカードにマッチするファイル名を見つけた場合、元のコマンドの引数をワイルドカードにマッチしたファイル名に置き換えます。
引数が置き換えられたコマンドを parse 関数に渡し、実行します。

6.4　テスト

以下はワイルドカードの実行結果である。cp * [ディレクトリ名] を入力し、カレントディレクトリにあるすべてのファイルを”*”で表した。また、ls -r コマンドを前後で実行することで、A_dir ディレクトリがもともと空であることを示した。

```
Command : ls -r A_dir
Command : cp * A_dir
Command : ls -r A_dir
script.txt mysh.c main.c a.out
replit.nix mysh B.txt
perfect.c Makefile A.txt
```

以上よりワイルドカード機能が正常に動作していることがわかる。

7　スクリプト機能

7.1　仕様

- ユーザがスクリプトファイルを作成し、シェルに渡すことでスクリプト内のコマンドを順番に実行する。
- スクリプトファイル内では、1 行に 1 つのコマンドを記述する。

7.2　処理の流れ (処理概要)

1. ユーザがスクリプトファイルを指定してシェルを起動する。、1 行ずつコマンドを解析・実行する。
2. スクリプト内の各コマンドが順番に実行される。
3. スクリプトファイルの最後まで実行が終わると、シェルが終了する。

7.3　実装方法

スクリプト機能は、シェルを起動する際にスクリプトファイル名を引数として受け取ります。
メインの無限ループ内で、スクリプトファイルを読み込み、1 行ずつコマンドを解析・実行します。
スクリプトファイルは fopen 関数を使用してファイルを開き、fgets 関数を使用して 1 行ずつ読み込みます。
読み込んだ行を parse 関数に渡し、コマンドと引数に分解します。
分解したコマンドと引数を execute_command 関数に渡し、コマンドを実行します。

7.4　テスト

script.txt に以下のコマンドを記述した。

```
echo "Hello"
pwd
ls
```

以下に./a.out < script.txt を実行した結果を示す。

```
Command : echo ”Hello”
”Hello”
Command : pwd
/home/runner/report2-u262691h
Command : ls
A_dir A.txt main.c mysh perfect.c script.txt
a.out B.txt Makefile mysh.c replit.nix
done.
```

以上からスクリプト機能が正常に動作していることがわかる。

8　エイリアス機能

8.1　仕様

- エイリアスはユーザーが定義できるコマンドの別名であり、コマンド名とエイリアスの対応を保存する機能を提供する。
- エイリアスは、コマンド名を指定するときに先頭に「!」を付けて表現することで使用できる。
- エイリアスの登録、一覧表示、削除が可能である。

8.2　処理の流れ (処理概要)

1. ユーザーが「alias」というコマンドを入力してエイリアス機能を呼び出す。
2. 引数がない場合、登録済みのエイリアス一覧を表示する。
3. 引数がある場合、新しいエイリアスを登録するか、既存のエイリアスを削除する。
4. エイリアスの登録は、コマンド名とエイリアスの対応を構造体配列に格納することで行う。
5. コマンド実行時にエイリアスが使用された場合、エイリアスをコマンド名に変換して実行する。

8.3　実装方法

typedef を使用してエイリアス構造体を定義する。
Alias 型の配列を定義し、エイリアスの対応を格納する。
alias() 関数では、エイリアスの登録を strcpy を用いて構造体の command1 と 2 にそれぞれコピーを行う。登録できたら alias_count を 1 増やす。
unalias() 関数では、登録されたエイリアスを削除するため、そのコマンドより後ろにあるコマンドを一つずつ前に上書きコピーしていき、alias_count を 1 減らす。
alias_count が 1 以上のときすなわち何かしらコマンドが登録されているとき、main 関数で command1 と args[0] が一致するものがないか for 文で探す。あったら、command2 のほうを args[0] に代入して実行される。

8.4　テスト

以下は alias の登録の実行結果である。

```
Command : alias dwp pwd
Command : dwp
/home/runner/report2-u262691h
```

以下は alias の一覧表示の実行結果である。

```
Command : alias dwp pwd
Command : alias sl ls
Command : alias
dwp pwd
sl ls
```

以下は unalias の実行結果である。

```
Command : alias dwp pwd
Command : dwp
/home/runner/report2-u262691h
Command : unalias dwp
Command : dwp
dwp: fatal error: no output file specified
```

以上よりエイリアス機能が正常に動作していることがわかる。

9 プロンプト変更機能

9.1 仕様

prompt コマンドの後に新しいプロンプト文字列を入力することで、その文字列がプロンプトとして表示される。

9.2 処理の流れ (処理概要)

1. ユーザーがコマンドラインに prompt コマンドを入力する。
2. parse 関数を使って、入力されたコマンドを解析し、args 配列に格納する。
3. execute_command 関数を使って、args 配列に格納されたコマンドを実行する。
4. prompt 関数を呼び出し、新しいプロンプト文字列を設定する。

9.3 実装方法

この関数は、args 配列に格納された新しいプロンプト文字列を受け取り、prompt_string 変数にその文字列をコピーする。
もし args が NULL の場合は、デフォルトのプロンプト文字列を設定する。

9.4 テスト

以下は prompt の実行結果である。

```
Command : prompt P
P :
```

以上よりプロンプトの変更機能が正常に動作していることがわかる。

10 オリジナル機能

10.1 仕様

ブラックジャック機能の仕様は、ユーザーがブラックジャックゲームをプレイできるようにすることである。ブラックジャックは、プレイヤーとディーラーがカードを引いて手札を作り、手札の合計値が 21 に近い方が勝利するカードゲームである。具体的なルールは以下の通りである：ゲーム開始時に、ディーラーとプレイヤーに 2 枚のカードが配られる。(このときユーザーは自分の手札 2 枚に加えディーラーの手札 1 枚を見ることができる。) カードの数字は 1 から 13 までの値を持ち、1 は「A」として、10 以上の数値はすべて 10 として扱われる。プレイヤーは「ヒット」または「スタンド」を選択することができる。ヒット：追加のカードを引く。スタンド：手札の合計値を確定し、ディーラーのターンへ移る。プレイヤーがスタンドを選択すると、ディーラーは 17 以上の手札合計値になるまでカードを引き続ける。手札の合計値が 21 に近い方が勝利し、21 を超えると「バースト」して負けとなる。(ただし、ディーラーとプレイヤーのどちらもバーストした場合はディーラー側の勝利となる。)

10.2 処理の流れ (処理概要)

1. ゲーム開始時に、ディーラーとプレイヤーに 2 枚のカードをランダムに配布する。
2. プレイヤーがカードを引くか、スタンドするかの選択を求める。
3. プレイヤーがヒットを選択した場合、ランダムに 1 枚のカードを引く。
4. プレイヤーの手札合計値が 21 を超えたらバーストとしてゲーム終了する。
5. プレイヤーがスタンドを選択した場合、ディーラーのターンへ移る。
6. ディーラーは手札合計値が 17 以上になるまでカードを引き続ける。
7. ディーラーの手札合計値が 21 を超えたらバーストとしてゲーム終了する。
8. ディーラーとプレイヤーの手札合計値を比較し、勝敗を判定する。

10.3 実装方法

このブラックジャック機能は 3 つの関数 (blackjack,convert_value_to_string,convert_value_to_10_or_less) で成り立っている。主要な処理を行うのが blackjack 関数で、convert_value_to_string 関数と convert_value_to_10_or_less 関数は、カードの表示や合計値を計算する際に用いる。

10.3.1 用いた変数・配列名

ブラックジャック機能で用いた変数や配列名について以下に表にまとめる。

変数名	型	用途
Dealer,Player	int	手札を格納する配列
dealerTotal,playerTotal	int	手札の合計値を格納する変数
numOfCardsDealtDealer,numOfCardsDealtPlayer	int	手札の枚数をカウントする変数
choice	char	プレイヤーの文字入力を格納する変数

10.3.2 補助関数の役割

補助関数の役割について説明する。
convert_value_to_string 関数は、受け取った変数のカードの文字を返す関数である。配列 card_values には、A から K までのカードが文字として格納され、受け取った数のカードの文字を返す。
convert_value_to_10_or_less 関数は、10 以上の値が渡されたとき、10 を返す関数である。これはブラックジャックでは絵札のカードは 10 としてカウントするためである。

10.3.3 手札の配布

ディーラーとプレイヤーの手札を管理するために、Dealer と Player という 2 つの配列が用意されている。ディーラーとプレイヤーの手札の合計値を格納するために、dealerTotal と playerTotal という 2 つの変数が用意されている。
ゲームの開始時には、ランダムに 2 枚のカードがディーラーとプレイヤーに配布されるため、それぞれの配列に srand 関数で生成した 1 から 13 までの乱数を２つずつ格納する。(time.h を include しておいた)
そして、その２枚の合計値を計算するため、convert_to_10_or_less 関数を通した値を dealerTotal,playerTotal に加算する。
printf で dealer の１枚目と Player の手札２枚を出力する。

10.3.4 Player のアクション

プレイヤーのターンでは、ヒット（カードを追加する）かスタンド（そのまま）かを選択できる。printf で、プレイヤーに h または s を入力するように促す。h を入力つまりヒットを選択した場合、新しいカードが 1 枚配布され、合計値が更新される。すなわち再び rand 関数で乱数を生成し、配列 player に格納し playertotal に加算して numOfCardsDealtplayer を１増やす。ただし、合計値が 21 を超えると、プレイヤーはバーストして負けとなるため、バーストと出力しループを抜ける。21 以下だった場合、ループして再び文字入力を促す。
s を入力つまりスタンドを選択した場合、ディーラーのターンにうつるため A の補正を行ってからループを抜ける。A の補正というのは、ブラックジャックでは A を１としても１１としても扱うことが可能であるため、for と if 文で配列 Player に A があるか判定し、A をもっていてかつ合計値が１１以下である場合は、playerTotal に１０を追加する。最後に、プレイヤーの合計値を出力してループを抜ける。

10.3.5 Dealer のターン

プレイヤーのターンが終了したら、ディーラーのターンに移る。ディーラーは合計値が 17 以上になるまでカードを追加する。そのため、while 文で dealerTotal が 17 未満である間は、乱数を一つ追加して dealerTotal に加算し、numOfCardsDealtDealer を１増やすという処理をループする。

10.3.6 勝負の判定

二者の手札を for 文を用いてそれぞれ numOfCardsDealt 枚ずつ出力する。次に、playerTotal が２１より大きいすなわちバーストしているとき、プレイヤーの負けであることを出力する。次に、dealerTotal が２１より大きいとき、ディーラーがバーストしたためプレイヤーの勝利であることを伝える。これは、実際にブラックジャックのルールでディーラーがバーストしていてもすでにプレイヤーがバーストしている場合はプレイヤーの敗北となるためこの順で判定を行った。あとはどちらもバーストしていないため dealerTotal と playerTotal を比較して大きいほうが勝利したことを出力する。

10.4 テスト

以下はコマンド bj の実行結果である。結果は様々な場合があるため、３回行った。

```
Command : bj
—[BLACK JACK]— GAME START!!
Dealer’s hands: J, ?
Player’s hands: 10, Q

INPUT ’h’ or ’s’（h=Hit, s=Stand）: s

Player Total: 20

[Result]
Dealer’s hands: J 10
Player’s hands: 10 Q
—Draw—
```

```
Command : bj
—[BLACK JACK]— GAME START!!
Dealer’s hands: 3, ?
Player’s hands: 4, 8

INPUT ’h’ or ’s’（h=Hit, s=Stand）: h

Player’s hands: 4 8 3

INPUT ’h’ or ’s’（h=Hit, s=Stand）: h

Player’s hands: 4 8 3 J
Burst!

[Result]
Dealer’s hands: 3 2 A 2
Player’s hands: 4 8 3 J
—Burst! Player Lose...—
```

```
Command : bj
—[BLACK JACK]— GAME START!!
Dealer's hands: Q, ?
Player's hands: A, 2

INPUT 'h' or 's'（h=Hit, s=Stand）: h

Player's hands: A 2 A

INPUT 'h' or 's'（h=Hit, s=Stand）: h

Player's hands: A 2 A 9

INPUT 'h' or 's'（h=Hit, s=Stand）: h

Player's hands: A 2 A 9 8

INPUT 'h' or 's'（h=Hit, s=Stand）: s

Player Total: 21

[Result]
Dealer's hands: Q 2 A 2 A 9
Player's hands: A 2 A 9 8
—Dealer Burst! Player Win!—
```

11　工夫点

ブラックジャックの実装が最も時間かかったが、その中で最も考えたのは、実際のカードに書かれている数とブラックジャックにおけるそのカードの数の意味が異なっていることだ。ブラックジャックにおいて A は 1 としても扱えるし、11 として扱えることができる。結果としては、はじめは A を 1 としてカウントして、total が 11 以下のとき 10 足すというシンプルなアルゴリズムでできた。

12　考察

今回自分はコマンドごとに関数として定義したが、一つ一つの関数の中身はそこまで長くないので、strcmp 関数で判定した直後にそのまま処理を書いちゃったほうがよりコンパクトにまとめることができたのかなと思った。

13　感想

めちゃくちゃ難しかったけど、今回の課題でスタックやヒストリーの格納などのＣ言語の基本的なところはほぼマスターできたと思うし、ディレクトリの操作など初めて知った関数もあったから勉強になった。トランプゲームが好きだからブラックジャックを実装している間はちょっと楽しくなってきてやりこみたくなったけど、シェルの実装はもうやりたくない...

14　謝辞

先生方や、特に授業中何度も質問させていただいた TA さん方、前期の間ありがとうございました。後期のプログラミング D も頑張ります。

15　プログラムリスト

```
1 /*------------------------------------------------------------------------
2  *  簡易版シェル
3  *------------------------------------------------------------------------*/
4
5 /*
6  *  インクルードファイル
7  */
8 #include <stdio.h>
9 #include <stdlib.h>
10 #include <string.h>
11 #include <sys/types.h>
12 #include <sys/wait.h>
13 #include <unistd.h>
14 #include <dirent.h>
15 #include <time.h>
16
17 /*
18  *  定数の定義
19  */
20
21 #define BUFLEN 1024   /* コマンド用のバッファの大きさ */
22 #define MAXARGNUM 256 /* 最大の引数の数 */
23 #define STACK_SIZE 10
24 #define PAST_SIZE 32
25 #define ALIAS_SIZE 10
26 /*
27  *  ローカルプロトタイプ宣言
28  */
29
30 int parse(char[], char *[]);
31 void execute_command(char *[], int);
32 void change_directory(char *args);
33 void push_directory();
34 void print_directory_stack();
35 void pop_directory();
36 void history();
37 void exclamation();
38 void string(char *args);
39 void prompt(char *args);
40 void alias(char *args[]);
41 void unalias(char *args);
```

```
42 void wild(char *args[]);
43 int blackjack();
44
45 typedef struct {
46     char command1[BUFLEN];
47     char command2[BUFLEN];
48 } Alias;
49
50 char *directory_stack[STACK_SIZE]; /* ディレクトリスタック */
51 int stack_index = 0;
52 char past_command[PAST_SIZE][BUFLEN];
53 int past_index = 0;
54 char prompt_string[BUFLEN] = "Command";
55 Alias alias_command[ALIAS_SIZE];
56 int alias_count = 0;
57 /*--------------------------------------------------------------------------
58  *
59  *  関数名    : main
60  *
61  *  作業内容 : シェルのプロンプトを実現する
62  *
63  *  引数      :
64  *
65  *  返り値    :
66  *
67  *  注意      :
68  *
69  *--------------------------------------------------------------------------*/
70
71 int main(int argc, char *argv[]) {
72   char command_buffer[BUFLEN]; /* コマンド用のバッファ */
73   char *args[MAXARGNUM];       /* 引数へのポインタの配列 */
74   int command_status;
75   int script_flag=0;
76   int eof_flag=0;
77   /* コマンドの状態を表す
78
79                                    command_status = 0 : フォアグラウンドで実行
80                                    command_status = 1 : バックグラウンドで実行
81                                    command_status = 2 : シェルの終了
82                                    command_status = 3 : 何もしない */
83
84   /*
85    *  無限にループする
86    */
87   if(isatty(fileno(stdin))==0){
88     script_flag = 1;
89   }
90   for (;;) {
91
92     /*
93      *  プロンプトを表示する
94      */
95
96     /*
97      *  標準入力から１行を command_buffer へ読み込む
98      *  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
99      */
100    if(eof_flag == 0){
101        printf("%s : ",prompt_string);
102    }
103
```

```
104     if(fgets(command_buffer,BUFLEN,stdin)==NULL){
105       command_status = 2;
106     }else{
107       if(script_flag == 1){
108         if(feof(stdin)!=0){
109           strcat(command_buffer,"\n");
110           eof_flag=1;
111         }
112         printf("%s",command_buffer);
113       }
114       //ヒストリーに保存
115       if(past_index < PAST_SIZE){
116         strcpy(past_command[past_index], command_buffer);
117       }else{
118         for(int i=0; i < PAST_SIZE-1; i++){
119           strcpy(past_command[i], past_command[i+1]);
120         }
121         strcpy(past_command[PAST_SIZE-1], command_buffer);
122       }
123       past_index++;
124       if(command_buffer[0] ==’!’ && command_buffer[1] != ’!’){
125       int k = strlen(command_buffer);
126       command_buffer[k+1]=’\0’;
127       for(int x=k-1;x>0;x--){
128         command_buffer[x+1]=command_buffer[x];
129       }
130       command_buffer[1]=’ ’;
131       }
132       command_status = parse(command_buffer, args);
133     }
134
135     if(alias_count>0){
136       for(int j=0;j<alias_count;j++){
137         if(strcmp(alias_command[j].command1, args[0])==0){
138         args[0]=alias_command[j].command2;
139         }
140       }
141     }
142
143     /*
144      *  終了コマンドならばプログラムを終了
145      *  引数が何もなければプロンプト表示へ戻る
146      */
147     if (command_status == 2) {
148       printf("done.\n");
149       exit(EXIT_SUCCESS);
150     } else if (command_status == 3) {
151       continue;
152     }
153         execute_command(args, command_status);
154   }
155   return 0;
156 }
157
158 /*------------------------------------------------------------------------
159  *
160  *  関数名    : parse
161  *
162  *  作業内容 : バッファ内のコマンドと引数を解析する
163  *
164  *  引数      :
165  *
```

```
166  *  返り値    : コマンドの状態を表す :
167  *                  0 : フォアグラウンドで実行
168  *                  1 : バックグラウンドで実行
169  *                  2 : シェルの終了
170  *                  3 : 何もしない
171  *
172  *  注意      :
173  *
174  *--------------------------------------------------------------------------*/
175
176 int parse(char buffer[], /* バッファ */
177           char *args[])  /* 引数へのポインタ配列 */
178 {
179   int arg_index; /* 引数用のインデックス */
180   int status;    /* コマンドの状態を表す */
181
182   /*
183    *  変数の初期化
184    */
185
186   arg_index = 0;
187   status = 0;
188
189   /*
190    *  バッファ内の最後にある改行をヌル文字へ変更
191    */
192
193   *(buffer + (strlen(buffer) - 1)) = '\0';
194
195   /*
196    *  バッファが終了を表すコマンド（"exit"）ならば
197    *  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
198    */
199
200   if (strcmp(buffer, "exit") == 0) {
201
202     status = 2;
203     return status;
204   }
205
206   /*
207    *  バッファ内の文字がなくなるまで繰り返す
208    *  （ヌル文字が出てくるまで繰り返す）
209    */
210
211   while (*buffer != '\0') {
212
213     /*
214      *  空白類（空白とタブ）をヌル文字に置き換える
215      *  これによってバッファ内の各引数が分割される
216      */
217
218     while (*buffer == ' ' || *buffer == '\t') {
219       *(buffer++) = '\0';
220     }
221
222     /*
223      * 空白の後が終端文字であればループを抜ける
224      */
225
226     if (*buffer == '\0') {
227       break;
```

```
228     }
229
230     /*
231      *  空白部分は読み飛ばされたはず
232      *  buffer は現在は arg_index + 1 個めの引数の先頭を指している
233      *
234      *  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
235      */
236
237     args[arg_index] = buffer;
238     ++arg_index;
239
240     /*
241      *  引数部分を読み飛ばす
242      *  （ヌル文字でも空白類でもない場合に読み進める）
243      */
244
245     while ((*buffer != '\0') && (*buffer != ' ') && (*buffer != '\t')) {
246       ++buffer;
247     }
248   }
249
250   /*
251    *  最後の引数の次にはヌルへのポインタを格納する
252    */
253
254   args[arg_index] = NULL;
255
256   /*
257    *  最後の引数をチェックして "&" ならば
258    *
259    *  "&" を引数から削る
260    *  コマンドの状態を表す status に 1 を設定する
261    *
262    *  そうでなければ status に 0 を設定する
263    */
264
265   if (arg_index > 0 && strcmp(args[arg_index - 1], "&") == 0) {
266
267     --arg_index;
268     args[arg_index] = '\0';
269     status = 1;
270
271   } else {
272
273     status = 0;
274   }
275
276   /*
277    *  引数が何もなかった場合
278    */
279
280   if (arg_index == 0) {
281     status = 3;
282   }
283
284   /*
285    *  コマンドの状態を返す
286    */
287
288   return status;
289 }
```

```
290
291 /*--------------------------------------------------------------------------
292  *
293  *  関数名    : execute_command
294  *
295  *  作業内容 : 引数として与えられたコマンドを実行する
296  *              コマンドの状態がフォアグラウンドならば、コマンドを
297  *              実行している子プロセスの終了を待つ
298  *              バックグラウンドならば子プロセスの終了を待たずに
299  *              main 関数に返る（プロンプト表示に戻る）
300  *
301  *  引数      :
302  *
303  *  返り値    :
304  *
305  *  注意      :
306  *
307  *--------------------------------------------------------------------------*/
308
309 void execute_command(char *args[], int command_status)
310 {
311   char *extracted_string = args[0] + 1;
312   for(int i=1;args[i]!=NULL;i++){
313     if(strcmp(args[i], "*") == 0){
314       wild(args);
315       break;
316     }
317   }
318
319   if (strcmp(args[0], "cd") == 0){
320         change_directory(args[1]);
321     }else if (strcmp(args[0], "pushd") == 0) {
322         push_directory();
323     } else if (strcmp(args[0], "dirs") == 0) {
324         print_directory_stack();
325     } else if (strcmp(args[0], "popd") == 0) {
326         pop_directory();
327     } else if (strcmp(args[0], "history") == 0){
328         history();
329     } else if (strcmp(args[0], "!!") == 0){
330         exclamation();
331     } else if(strcmp(args[0], "!") == 0){
332         string(args[1]);
333     } else if(strcmp(args[0], "prompt") == 0){
334         prompt(args[1]);
335     } else if(strcmp(args[0], "alias") == 0){
336         alias(args);
337     } else if(strcmp(args[0], "unalias") == 0){
338         unalias(args[1]);
339     } else if(strcmp(args[0], "bj")==0){
340         blackjack();
341     } else{
342       pid_t pid;   /* プロセス ID */
343       int status;  /* 子プロセスの終了ステータス */
344
345       pid = fork();  /* 新しいプロセスを生成 */
346
347       if (pid < 0) {
348         perror("fork");
349         exit(EXIT_FAILURE);
350       } else if (pid == 0) {
351         /* 子プロセスの場合 */
```

```
352
353         /* 引数として与えられたコマンドを実行 */
354         if (execvp(args[0], args) == -1) {
355             perror("execvp");
356             exit(EXIT_FAILURE);
357         }
358
359         /* execvp が成功すればここには到達しない */
360       } else {
361         /* 親プロセスの場合 */
362
363         if (command_status == 0) {
364             /* コマンドの状態がフォアグラウンドの場合、子プロセスの終了を待つ */
365             waitpid(pid, &status, 0);
366         } else {
367             /* コマンドの状態がバックグラウンドの場合、待たずにプロンプトに戻る */
368             printf("Background process started with PID %d\n", pid);
369         }
370     }
371   }
372   return;
373 }
374
375 void change_directory(char *args) {
376     int result;
377     if (args == NULL) {
378         /* 引数がない場合は親ディレクトリに移動する */
379         result = chdir("..");
380     } else {
381         /* 引数がある場合は指定されたディレクトリに移動する */
382         result = chdir(args);
383     }
384     if (result == -1) {
385         perror("chdir");
386     }
387     return;
388 }
389
390 void push_directory(const char *dir) {
391     if (stack_index >= STACK_SIZE) {
392         printf("エラー: ディレクトリスタックが一杯です。これ以上ディレクトリを追加できません。\n");
393         return;
394     }
395     char *current_dir = getcwd(NULL, 0);
396     if (current_dir == NULL) {
397         perror("getcwd");
398         return;
399     }
400     directory_stack[stack_index] = current_dir;
401     stack_index++;
402     return;
403 }
404
405 void print_directory_stack() {
406     if (stack_index == 0) {
407         printf("ディレクトリスタックは空です。\n");
408         return;
409     }
410
411     for (int i = stack_index - 1; i >=0; i--) {
412         printf("%s\n", directory_stack[i]);
413     }
```

```
414   return;
415 }
416
417 void pop_directory() {
418     if (stack_index == 0) {
419         printf("エラー: ディレクトリスタックは空です。ポップできません。\n");
420         return;
421     }
422     int result;
423     result = chdir(directory_stack[stack_index-1]);
424     if (result == -1) {
425         perror("chdir");
426     } else {
427         free(directory_stack[stack_index - 1]);
428         stack_index--;
429     }
430     return;
431 }
432
433 void history(){
434     for (int i = 0; i < past_index; i++) {
435         printf("%d: %s", i + 1, past_command[i]);
436     }
437   return;
438 }
439
440 void exclamation(){
441   if(past_index==0){
442     printf("過去に実行したコマンドはありません。\n");
443   }else{
444     char *last_command;
445     if(past_index < PAST_SIZE){
446       strcpy(past_command[past_index-1], past_command[past_index-2]);
447       last_command = past_command[past_index - 2];
448     }else{
449         for(int i=0; i < PAST_SIZE-1; i++){
450           strcpy(past_command[i], past_command[i+1]);
451         }
452         strcpy(past_command[PAST_SIZE-1], past_command[PAST_SIZE-2]);
453         last_command = past_command[PAST_SIZE - 1];
454     }
455     char *args2[MAXARGNUM];
456     int command_status = parse(last_command , args2);
457     execute_command(args2, command_status);
458   }
459   return;
460 }
461
462 void string(char *args){
463   if(past_index==0){
464     printf("過去に実行したコマンドはありません。\n");
465   }else{
466     int k =strlen(args);
467     char string_command[BUFLEN];
468     char *last_command;
469     for(int l=past_index;l>=0;l--){
470       if (strncmp(past_command[l], args, k)==0){
471         if(past_index < PAST_SIZE){
472           strcpy(past_command[past_index-1], past_command[l]);
473           last_command = past_command[l];
474         }else{
475           for(int i=0; i < PAST_SIZE-1; i++){
```

```
476             strcpy(past_command[i], past_command[i+1]);
477           }
478           strcpy(past_command[PAST_SIZE-1], past_command[PAST_SIZE-2]);
479           last_command = past_command[PAST_SIZE - 1];
480         }
481         char *args2[MAXARGNUM];
482         int command_status = parse(last_command , args2);
483         execute_command(args2, command_status);
484         return;
485       }
486     }
487     printf("その文字列で始まるコマンドはヒストリに格納されていません。");
488   }
489   return;
490 }
491
492 void prompt(char *args){
493   if (args != NULL) {
494     strcpy(prompt_string, args); // 引数があればプロンプト文字列を更新
495   } else {
496     strcpy(prompt_string, "Command"); // 引数がなければデフォルトに戻す
497   }
498 }
499
500 void alias(char *args[]){
501   if(args[1] == NULL || args[2] == NULL){
502     if(alias_count==0){
503       printf("登録されたコマンドはありません。\n");
504     }else{
505       for(int i=0; i < alias_count;i++){
506         printf("%s %s\n",alias_command[i].command1,alias_command[i].command2);
507       }
508     }
509   } else if (args[2] == NULL){
510     printf("エラー: 置き換えたいコマンド名を入力してください。");
511   } else {
512     strcpy(alias_command[alias_count].command1, args[1]);
513     strcpy(alias_command[alias_count].command2, args[2]);
514     alias_count++;
515   }
516 }
517
518 void unalias(char *args){
519   for(int i=0;i<alias_count;i++){
520     if(strcmp(alias_command[i].command1, args)==0){
521       for(int j = i; j < alias_count - 1; j++){
522         strcpy(alias_command[j].command1, alias_command[j + 1].command1);
523         strcpy(alias_command[j].command2, alias_command[j + 1].command2);
524       }
525         alias_count--;
526         return;
527     }
528   }
529   printf("その名前で登録したコマンドはありません。");
530 }
531
532 void wild(char *args[]) {
533   DIR *dir;
534   struct dirent *entry;
535   char file[BUFLEN] = "\0";
536   char new_args[BUFLEN] = "\0";
537   if ((dir = opendir(".")) == NULL) {
```

```
538     perror("opendir");
539     return;
540   }
541   while ((entry = readdir(dir)) != NULL) {
542     if (entry->d_type == DT_REG) {
543       strcat(file, entry->d_name);
544       strcat(file, " ");
545     }
546   }
547   strcat(file, "\0");
548   for (int k = 0; args[k] != NULL; k++) {
549     if (strstr(args[k], "*") !=
550         NULL) {
551       strcat(new_args, file);
552     } else {
553       strcat(new_args, args[k]);
554       strcat(new_args, " ");
555     }
556   }
557   strcat(new_args, "\0");
558   int i = parse(new_args, args);
559   if (closedir(dir)) {
560     perror("closedir");
561     return;
562   }
563   printf("\n");
564   return;
565 }
566
567 // 数値を J, Q, K に変換する関数
568 const char* convert_value_to_string(int value) {
569     static const char* card_values[] = { "A", "2", "3", "4", "5", "6", "7", "8", "9", "10", "J
570     return card_values[value - 1];
571 }
572
573 // 数値を 10 以下の値に変換する関数（11 以上は 10 として扱う）
574 int convert_to_10_or_less(int value) {
575     return (value > 10) ? 10 : value;
576 }
577
578 int blackjack() {
579     // 乱数のシード値を設定する（通常、プログラムの先頭で一度だけ行う）
580     srand(time(NULL));
581
582     int Dealer[5]; // Dealer の手札（最大 5 枚）を格納する配列
583     int Player[5]; // Player の手札（最大 5 枚）を格納する配列
584     int dealerTotal = 0; // Dealer の手札の合計値
585     int playerTotal = 0; // Player の手札の合計値
586     int numOfCardsDealtDealer = 2; // Dealer の手札枚数
587     int numOfCardsDealtPlayer = 2; // Player の手札枚数
588
589     // 初期手札の配布
590     for (int i = 0; i < numOfCardsDealtDealer; i++) {
591         Dealer[i] = rand() % 13 + 1;
592         dealerTotal += convert_to_10_or_less(Dealer[i]);
593     }
594
595     for (int i = 0; i < numOfCardsDealtPlayer; i++) {
596         Player[i] = rand() % 13 + 1;
597         playerTotal += convert_to_10_or_less(Player[i]);
598     }
599     printf("---[BLACK JACK]---   GAME START!!\n");
```

```
600     // 初期手札の表示
601     printf("Dealer's hands: %s, ?\n", convert_value_to_string(Dealer[0]));
602     printf("Player's hands: %s, %s\n", convert_value_to_string(Player[0]), convert_value_to_st
603
604     // プレイヤーのターン
605     while (1) {
606         char choice;
607         printf("\nINPUT 'h' or 's' (h=Hit, s=Stand) : ");
608         int num_items_read = scanf(" %c", &choice);
609         getchar(); // 改行文字をクリア
610         if (num_items_read != 1) {
611     // エラー処理
612         printf("入力エラー\n");
613     // 必要に応じて処理を中止するなどのエラーハンドリングを行う
614         break;
615         }
616
617         if (choice == 'h' || choice == 'H') {
618             // Hit の場合、Player に 1 つ乱数を追加
619             Player[numOfCardsDealtPlayer] = rand() % 13 + 1;
620             printf("\nPlayer's hands: ");
621             for (int i = 0; i <= numOfCardsDealtPlayer; i++) {
622                 printf("%s ", convert_value_to_string(Player[i]));
623             }
624             printf("\n");
625
626             playerTotal += convert_to_10_or_less(Player[numOfCardsDealtPlayer]);
627             numOfCardsDealtPlayer++;
628
629             // バーストの判定
630             if (playerTotal > 21) {
631                 printf("Burst!\n");
632                 break;
633             }
634         } else if (choice == 's' || choice == 'S') {
635             // Stand の場合、ディーラーのターンへ
636             // A を 11 として扱った場合の処理
637             int numOfAces = 0;
638             playerTotal = 0;
639             for (int i = 0; i < numOfCardsDealtPlayer; i++) {
640               if (Player[i] == 1) {
641                 numOfAces++;
642               }
643             playerTotal += convert_to_10_or_less(Player[i]);
644             }
645
646             if (numOfAces > 0 && playerTotal + 10 <= 21) {
647               playerTotal += 10;
648             }
649             printf("\nPlayer Total: %d\n", playerTotal);
650             break;
651         }
652     }
653
654     // ディーラーのターン
655     while (dealerTotal < 17) {
656         // Dealer に 1 つ乱数を追加
657         Dealer[numOfCardsDealtDealer] = rand() % 13 + 1;
658         dealerTotal += convert_to_10_or_less(Dealer[numOfCardsDealtDealer]);
659         numOfCardsDealtDealer++;
660
661         // A を 11 として扱った場合の処理
```

```
662         if (Dealer[numOfCardsDealtDealer - 1] == 1 && dealerTotal <= 11) {
663             dealerTotal += 10;
664         }
665     }
666
667     // 勝敗の判定
668     printf("\n[Result]\n");
669     printf("Dealer's hands: ");
670     for (int i = 0; i < numOfCardsDealtDealer; i++) {
671         printf("%s ", convert_value_to_string(Dealer[i]));
672     }
673     printf("\n");
674     printf("Player's hands: ");
675     for (int i = 0; i < numOfCardsDealtPlayer; i++) {
676         printf("%s ", convert_value_to_string(Player[i]));
677     }
678     printf("\n");
679
680     if (playerTotal > 21) {
681         // プレイヤーがバーストしている場合
682         printf("---Burst! Player Lose...---\n");
683     } else if (dealerTotal > 21) {
684         // ディーラーがバーストしている場合
685         printf("---Dealer Burst! Player Win!---\n");
686     } else if (playerTotal == dealerTotal) {
687         // 引き分けの場合
688         printf("---Draw---\n");
689     } else if (playerTotal > dealerTotal) {
690         // プレイヤーの手札がディーラーの手札よりも大きい場合
691         printf("---Player Win!---\n");
692     } else {
693         // ディーラーの手札がプレイヤーの手札よりも大きい場合
694         printf("---Player Lose...---\n");
695     }
696 }
697 /*-- END OF FILE ------------------------------------------------------------*/
```