1　課題 5

1.1　サーバーのみ実行時

1.2　クライアント実行時

2　課題 6

2.1　common.c

2.1.1　copy_table()

■何の処理？

第二引数の配列の任意の値を第一引数の配列へコピーする処理。

■配列への処理内容

第二引数の配列の x 行 y 列の値を第一引数の同じ場所へ代入する

■機能実現への決め手

for 文を 2 重にネストし、dst_cards[i][j]=src_cards[i][j] とすることで実現が可能となった。

2.1.2　clear_table()

■何の処理？

引数として受け取ったカードテーブルを初期化する。

■配列への処理内容

配列のすべての値を 0 にする。

■機能実現への決め手

for 文を 2 重にネストし、cards[i][j]=0 とすることで実現が可能となった。

2.1.3　copy_cards()

■何の処理？

第二引数の配列のカード情報を第一引数の配列へコピーする処理。

■配列への処理内容

第二引数の配列の x 行 y 列 (x ≦ 5) の値を第一引数の同じ場所へ代入する

■機能実現への決め手

基本的には copy_table() と処理内容は同じである。1 点のみ違いがあり、for 文でネストする際に i の制限を 8 ではなく 5 とすることで、カード情報のみ指定して初期化することが可能となった。

2.1.4　clear_cards()

■何の処理？

引数で受け取ったカードテーブルのカード情報の部分のみ初期化する。

■配列への処理内容

カードテーブルの第 5 行までに限り (列は任意)、配列の値を 0 にする。

■機能実現への決め手

基本的には clear_table() と処理内容は同じである。1 点のみ違いがあり、for 文でネストする際に i の制限を 8 ではなく 5 とすることで、カード情報のみ指定して初期化することが可能となった。

2.1.5 diff_cards()

■何の処理？

ゲーム開始前の貧民・大貧民→大富豪・富豪のカード譲渡において、引数で指定された (search_low_card() にて指定された貧民・大貧民持ち札で最も強い 1-2 枚の) カードを貧民・大貧民の持ち札から削除する処理。

■配列への処理内容

第一引数の自分の持ち札を表す cards1 と、事前に search_low_card() にて譲渡=削除すべきカードを cards2[x][y]=1 とされた配列を第二引数として受け取った。cards1 配列において、cards2[x][y] にて 1 とされた場所を 0 にする、つまり cards1[x][y] に 0 を代入すること。

■機能実現への決め手

for 文を 2 重にネストすることで 2 次元配列 cards1[][] 全体を探索し、if 文として 1 と指定された場所に限り、cards1 の同一の行・列部分に 0 を代入することで実現可能となった。

2.1.6 or_cards()

■何の処理？

ゲーム開始前の大富豪・富豪→貧民・大貧民のカード譲渡において、引数で指定された (search_low_card() にて指定された貧民・大貧民持ち札で最も強い 1-2 枚の) カードを富豪・大富豪の持ち札へ追加する処理。

■配列への処理内容

第一引数の自分の持ち札を表す cards1 と、事前に search_low_card() にて譲渡=削除すべきカードを cards2[x][y]=1 とされた配列を第二引数として受け取った。cards1 配列において、cards2[x][y] にて 1 とされた場所を 1 にする、つまり cards1[x][y] に 1 を代入すること。

■機能実現への決め手

for 文を 2 重にネストすることで 2 次元配列 cards1[][] 全体を探索し、if 文として 1 以上に指定された場所に限り、cards1 の同一の行・列部分に 1 を代入することで実現可能となった。

2.1.7 and_cards()

■何の処理？

第一引数の cards1 配列において、cards2 においても 1 として指定された場所以外をすべて 0 にする処理。

■配列への処理内容

cards1[][] において、cards1 と cards2 が同じ場所の配列の値がともに 1 である場合に限り、cards1 の該当値を 1 のままとする。両方共 1 でないのならば、cards1 の該当値は 0 が代入される。

■機能実現への決め手

for 文を 2 重にネストすることで 2 次元配列 cards1[][] 全体を探索し、if 文として cards1[i][j] と cards2[i][j] がともに 1 である場合を&&で指定し、その場合は cards[j][i] に 1 を代入し、それ以外の場合は else として cards[j][i] に 0 を代入することで、実現可能となった。

2.1.8 not_cards()

■何の処理？

カードの有無を配列の値を 0 と 1 とで入れ替えることで反転する処理。

■配列への処理内容

配列の任意の場所の値を 0 ⇄ 1 とで反転している。

■機能実現への決め手

for 文を 2 重にネストすることで 2 次元配列 cards[][] 全体を探索し、cards[j][i]=1 ならば 0 を、0 ならば 1 を cards[j][i] に代入することで、実現可能となった。

2.1.9 get_field_state_from_field_cards()

■何の処理？

場に出ているカードの情報から、現在の場の状況を把握する処理。

■配列への処理内容

初期段階判定　cards[i][j] が 1 である場所を捜し、存在しているカードのスートを field_status の suit[i] に記録する。すべてのスートを調べた (i=4) ならば j をインクリメントし、i を 0 に戻す。

階段判定　cards[i][j] において、cards[i][J+1] も場に出ている=1 である場合、階段が発動していると判定し、field_status の is_sequence に 1 を代入する。

枚数組判定　階段が不成立の場合 (field_status=0)、まず同一数値における別スートのカードの有無を調べ、存在するならば枚数をカウントする count をインクリメントし、suit[i] に 1 を代入する。存在しなければ suit[i] を代入する。

革命有無判定　階段が不成立で j が 0 または 14 のとき、革命が発動中ならば、field_status の order に 14 を代入し、力関係が逆転していることを伝える。革命中でないのならば、order には 0 が代入される。j が 0 でも 14 でもない時は、j の値が order に代入される。

階段処理判定　階段が成立中の場合、出されたカードの最大値 (j)+1 セットの枚数 (count) が 15 を超えず、かつ cards[i][j+count] が 0 でない (場に出ている状態である) 間、while として count をインクリメントする。その後、革命中ならば field_status の order には j+count-1 を代入し、革命中でなければ j をそのまま代入する。最後に、field_status の suit[i] に 1 を代入する。

場カード有無　上述のコードによって判定された count を field_status の quantity に代入し、更に quantity の数値が 0 でないのならば、field_status の is_no_card に 0 を代入する。

■機能実現への決め手

初期段階判定　while で j を先に固定し、その後 i を 0 から 4 までインクリメントしている。スートよりも数字の値の方が重要性が高いため、先に固定することでカード位置の正確な探索が可能になった。
階段判定　もし場に同じスートで数字が隣接するカードも出ている場合を if-else で判別することで可能となった。
枚数組判定　j の値は変えずに、i のみ 0 から 4 までインクリメントして cards[i][j] の値を調べることにより、同一数値で何枚カードが存在するか調べることが可能になった。
革命有無判定　field_status に is_rev という革命有無を記す変数が搭載され、その値の有無によって if-else を分けることにより、可能なものとなった。
階段処理判定　先にスートを固定し、隣接するカードがなくなるまで while 文を回すことにより、セットで出された階段の枚数の判定が可能になった。また、革命の有無によって次に出すべき階段のセットの判定を変えることにより、正確に次に出すべきカードを判定することが可能になった。
場カード有無　count の値を quantity に移植、さらに quantity が 0 を超えることを if 文でキャッチしその場合 is_no_card に 0 を代入することで新場なのか否かを判別できるようになった。

2.1.10　get_field_state_from_own_cards()

■何の処理？
引数として受け取った手札のカードテーブル cards[8][15] から、field_status 所蔵の各状態メンバを操作する処理。

■配列への処理内容
場カード状況による操作　場にカードがなければ (cards[5][4] の値の有無)field_status の is_no_card に 1 を、革命状態 (cards[5][6] の値の有無) ならば is_rev に 1 を、縛り状態 (cards[5][7] の値の有無) なら is_lock に 1 を代入する。
新場状態の処理　新たな場となった場合、field_status 内の場に出されたカードセット数 quanity や強さを表す order、縛りを表す is_lock や出されたカードのスートを表す suit[] をすべて初期化する。
プレーヤーステータスの整理　cards[6][0-4] までが表す手持ちのカード状態を field_status の player_quantity[5] にそれぞれ代入する。その後、プレーヤーのランク (役職) が cards[6][5-9] までが表しているので、それぞれを player_rank[5] に代入する。最後に、試合ごとにシャッフルされる座席情報が cards[6][10-14] に記載されているので、それぞれを seat[5] に代入する。
Joker **所持未所持の処理**　cards[4][1] に 2 が代入されているのはプレーヤーは Joker を所持していることを表しているため、field_status の have_joker に 1 が代入される。

■機能実現への決め手
場カード状況による操作　cards[5][4, 6, 7] にそれぞれ場で発生している状態が一括で記載されており、その値を参照することで可能となった。
新場状態の処理　操作・参照する変数が field_status 内メンバで完結しており、わかりやすく処理を書かれることが可能となった。
プレーヤーステータスの整理　cards[6][1-14] にプレーヤーのステータスが一括で記載されており、それぞれを for 文で囲むことにより、処理をそれぞれ 1 行ずつで完結することが可能となった。

Joker 所持未所持の処理　cards[4][1] に Joker の有無を 2 を代入することで記載されており、if 文で該当値を取得・分岐することで可能となった。

2.1.11　remove_low_card()

■何の処理？

配列 cards[][] の指定されたエリアの値をすべて 0 とする処理。

■配列への処理内容

第一引数の cards[i][j] において、第二引数で指定された i 以上/以下 (j は任意) にて指定された部分を 0 とする。以上/以下は第三引数 rev(革命の有無) が 0 なら以下、1 ならば以上を採用する。

■機能実現への決め手

if 文で第三引数 rev の有無で処理を分け、それぞれで for 文を 2 重にネストして 0 とするエリアを網羅的に指定することで可能となった。

2.1.12　remove_suit()

■何の処理？

縛りが発動している際に、指定スート以外のカードを出せないように制限する処理。

■配列への処理内容

第一引数 cards[i][j] に対して、第二引数 suit[i] と第三引数 flag(0 か 1) の合計値が偶数ならば、該当する cards[i][j] の i 行中の任意の j 列の値を 0 とする。

■機能実現への決め手

for 文でまず suit[i] を調べ、flag との合計値が偶数であることを if 文と剰余演算子でキャッチする。該当する列を、ネストされた for 文で j を 0-14 まで動かして 0 を代入することで可能となった。

2.1.13　count_cards()

■何の処理？

受け取ったプレーヤーが所持しているカード枚数の合計を返す処理。

■配列への処理内容

第一引数として受け取った cards[i][j] において、i¡5, j 任意の条件下で 0 でない=所持しているカードがあれば、枚数をカウントする quantity 変数をインクリメントする。すべてのエリアを参照したら、quantity を返す。

■機能実現への決め手

for 文を 2 重にネストし、i の条件を 5 未満に制限することで純粋にカード所持情報を取得することが可能となった。

2.2 select_cards.c

2.2.1 select_change_cards()

■何の処理？

試合開始前に行われる貧民・大貧民→大富豪・富豪の最も弱い 1-2 枚のカードの譲渡を管理する処理。

■配列への処理内容

基本的にこの関数は直接配列への代入は行わず、先述の common.c 記載関数の組み合わせが行われている。第一引数 out_cards[][] は譲渡するカードを表すため、clear_table() を呼び出して初期化する。その後 out_cards[][] と第二引数 my_cards[][] に対して、交換する枚数が指定された第三引数 num_of_change の数だけ、search_low_card() と diff_cards() と or_cards() を繰り返して、my_cards[][] から out_cards[][] に譲渡する。

■機能実現への決め手

common.c にて作成した関数を組み合わせ、my_cards[][] から譲渡する out_cards[][] を作成することにより、可能なものとなった。

2.2.2 select_submit_cards()

■何の処理？

場に出すカードを選択する処理。

■配列への処理内容

まず、場に出すカードを表すカードテーブル select_cards[8][15] を定義し、clear_table() により初期化する。

その後、革命中 (field_status 内メンバ is_rev)・新たな場 (field_status 内メンバ is_no_card) によって場に出すカードを指定する関数を使い分けている。使用する関数は以下に表される。

なお、渡す引数は select_cards, my_cards, field_status と共通している。

表 1 使用する関数名

状態	通常: is_rev=0	革命: is_rev=1
新場: is_no_card=1	select_cards_free	select_cards_free_rev
既存場: is_no_card=0	select_cards_restrict	select_cards_restrict_rev

■機能実現への決め手

上記のように、field_status 内メンバによって呼び出す関数を使い分けることにより、革命・や場の状況に応じて最適なカードを出すことが可能となった。

2.2.3 select_cards_free()

■何の処理？

場にカードがなにもないときに呼び出され、手札で最も弱いカードを場に出す処理。

■配列への処理内容

daihinmin.c にて定義された search_low_card() 関数へ提出するカードを記した第一引数 select_cards[][] と、手札を記した第二引数 my_cards[][] と、Joker を使わないことを表す 0 をそれぞれ引数として渡している。

■機能実現への決め手

daihinmin.c の search_low_card へそのまま引数として受け取った配列を渡すことにより、実現可能となった。

2.2.4 select_cards_restrict()

■何の処理？

場にカードが既にあるときに呼び出され、更に場の状況に応じて出すことが可能なカードを場に出す処理。

■配列への処理内容

まず、場に出すことができるカードを記載するカードテーブル tmp_cards[][] 配列を定義し、copy_table 関数にて第二引数の手札のカードテーブル my_cards[][] 配列の内容をコピーする。

その次に if 文として第三引数 field_status 内メンバ is_sequence が 1 のときと、quantity が 1 のときをそれぞれ検知し、更に縛りの有無によって分岐が用意されているが、中身は実装されていない。

処理が実装されているのは場が単騎のみの場合で、この時に場が縛られている (field_status 内メンバ is_lock の値の有無) かを if 文で検知し分岐している。縛られているときは remove_suit 関数を先に入れて tmp_cards[][] を縛られたスートに制限する。その後は両者共通で remove_low_card() によって、tmp_cards[][] 内で場のカードより弱いカードを削除する。最後に、search_low_card() にて、場に出せられるカードが記載された tmp_cards[][] から最も弱いカードを指定し、第一引数 select_cards[][] に反映する。

■機能実現への決め手

まず場の状況が階段・ペア・単騎によって分け、更に縛られている・いない場合によって分けることで、それぞれの状況に応じた処理を実装することが可能となった。