知能情報工学実験演習 II

C++演習 (Part 1)

担当: 中村貞吾，下薗真一
TA: 石田竹至，蓑代成功
e-mail: {teigo, t_ishida, minoshiro } @dumbo.ai.kyutech.ac.jp,
sin@ai.kyutech.ac.jp

Apr. 2015

期間: 第 3 年次前期

機材: CAD 実験室 (共通教育研究棟 3 階 S303 室) の Linux PC 端末

注意: この実験・演習は第一部 Part 1 と第二部 Part 2 をそれぞれ 3 回，合計 6 回で行う．関連科目の復習ほか事前学習を十分行ったうえでのぞむこと．事前学習が不十分で時間が不足となった場合は，自宅等のＰＣや本実験・演習を行う計算機，その他研究棟側の端末などを活用することで補うこと．
なお，Part 1 の演習は個人単位で行ない，各自報告書を作成，提出すること．

1 はじめに（演習の目標）

C 言語および C++ 言語を使った実用的なプログラムに必要なプログラミング作法，つまり読みやすくミスの生じにくいプログラムの書き方を身につける．その中で，構造化プログラミングと，その発展形であるオブジェクト指向プログラミングの一部の理解を深め，活用できるようになることを目標とする．

まず簡単に,「プログラミング演習の課題をこなすためのプログラミング」から脱却するうえで有効な指針を挙げておく．

1.1 同じことなら単純なものほどよい（オッカムの剃刃）

プログラムを書くとき，あまり深く考えず，とりあえず必要そうな変数を山のように宣言するのをよく見かける．意味のわからない名前の変数や関数，使われない変数や引数がそのまま残っている．もし自分がそうなら，すぐやめること．

プログラムを書くのは，盛り込まれる規則，知識，アイディアを，共同開発者や利用者など，未来の自分自身をふくむ多くの人と共有し，活用するためである．そのためには，客観的な基準で整理整頓されている必要があり，これを実現する最も有効な方法は，単純で簡潔にすることなのである．

1.2 まず “データ型” を作る

整数型や文字型，つまりバイトやワードなど CPU が直接操作できる型だけで実用的なプログラムを作ることは，ハードウェアを意識せざるをえない組み込みマイコンのプログラムなどを除けば，めったにない．たいていは，人間にとって意味のあるデータの単位をプログラムで扱う「型」として表現することからはじめることになる．実はこの作業が，プログラムを単純で簡潔にするための鍵であり，最終的にプログラムを作成することで解決しようとしている問題の解決の鍵である．

C/C++ では，構造体もしくはクラスというデータ型でデータ構造をメモリのバイトやビットの単位で定義でき，さらにその操作を関数に対応させてプログラムを書くことができる．この特徴を生かしてこそ言語の価値があるといえる．

1.3 大域変数は禁止

大域変数はいつでもどこででも使え，引数に書かなくても関数の間でデータ受け渡しができるので即席プログラムには便利である．しかし，プログラムが大きく複雑になると，変数がどこでなんのため使われているかわからなくなる．とりわけ，オブジェクト指向プログラミングでは “大域変数を使いたい病” は大きな障害になる．

データを型として組み合わせてまとめること，使う変数を最少限にすることが，プログラムをうまく作るコツである．関数のデータのやり取りも，操作対象や動作を単純で明快なものにするため，引数と返り値のみで行うべきである．

2 C 言語での型づくり（復習）

2.1 データ型は，一定の意味を持たせたモデルと操作の組合せ

デジタル機器では，どんなデータもビットの列で表す．例えば，整数 `1040187392` と実数 `0.125F` はどちらも 32 ビットの列

```
0011 1110 0000 0000 0000 0000 0000 0000 (3E 00 00 00)
```

である．これはさらに，長さ 2 の文字列であるかもしれないし，音楽 CD に記録されたステレオ音声データの 1 サンプル目かもしれない．計算機のハードウェアは，どのアドレスのデータが何型かといった情報を持つための機構を用意していない．我々がプログラムの中で代入や演算をさ

まざまなデータ型に対して使用できるのは，int 型の変数やデータには int 型の操作 (命令) を，float 型の変数やデータには float 型の操作を，変数やデータの型にあわせてコンパイラが選んでいるからである．

データ型の本質は，内部表現の方法ではなく，**モデル** (例えば数学でいう「整数」) と**操作** (例えば整数の加算演算) の対応関係にある．実際に，int 型や float 型がメモリの中でどのように表されるかを知らなくても，我々はプログラムを書ける．データ型がどんな内部表現か知らなくても（普通は知らない），操作がその内部表現に対応していれば正しく動作するのだ．

C 言語のプログラムでは，対応づけは整数型（言語仕様で定義された型）に対してはコンパイラが行い，それ以外の型についてはプログラマが管理する．なお C++ では，さらに構造体／クラスなどプログラマが定義したデータ型と関数を対応させて書けるようになっている．

2.2 計算機中に存在しないものを表す

プログラムで必要となるデータ型には様々なものがありうるので，言語であらかじめそのすべてを用意することはできない．整数や浮動小数点数ですら，桁や精度の制限があり，数学的な整数や実数が表されているわけではない．必要なデータ型は，すでに使えるようになっている型を組み合わせて**自分で作るもの**なのである．これは**抽象データ型**とよばれる．

自分で型を作るとき，以下の二つが重要である．

(1) データ一個を表すのに必要なデータ（値）は，すべて一つにまとめる．
例えば，浮動小数点数では符号部と指数部と仮数部を一まとまりにして扱っている．一つなのであれば，別々にならないようにする．

(2) 一個のデータは，いつもまとめて操作する．
例えば，浮動小数点数の演算をすると，符号部も指数部も仮数部も一緒に使われ計算が行われる．その一部だけを操作して整合性が失われるようなことがないようにする．

なぜなら，人間にとって，一つと考えられるものは「一つ」として扱えるほうが楽で，わかりやすく，間違いがないからである．

2.3 構造体でデータを一まとめにする

一まとめにするには，必要なデータを組み合わせる．この目的のために，C 言語では，主に構造体 (struct) を宣言することになる．(C++言語では，構造体をクラス class ともよぶ．) 組み合わせに使うデータの型が言語で用意されたものか，定義ずみの抽象データ型かは，問われない．

例えば，数学でいう $\mathbb{R}^2$ の要素，2 次元ベクトル（のようなもの）を表すには，それぞれの成分を例えば double 型で表すことにして次のような構造体 vector2 を作ればよい．

```
struct vector2 {
        double x, y;        /* それぞれ x 成分，y 成分 */
};
```

また `double val[2];` のようにして，x, y の各値を配列の要素で表すこともできる．

2.4 データの操作は一まとめで行なえるようにする

データ型を作ったら，その中身を気にすることなく扱えるよう，操作を準備する．その型のデータそのものが，もともと言語にあるかのように扱う関数を定義する．そうすることにより，後で型の定義を変更することになったとしても — これは実際によく起きるが — 操作関数だけを変更すればよい．

例えば，2 次元ベクトルの加法演算は，

```
struct vector2 addv2(struct vector2 u, struct vector2 v)
{
        struct vector2 ret;

        ret.x = u.x + v.x;
        ret.y = u.y + v.y;

        return ret;
}
```

と用意し，プログラムで 2 次元ベクトル同士の加算が必要なら，

```
vec0 = addv2(vec1, vec2);
```

などと書く．このとき，関数の名前は対象となる型の名前と操作の名前を組み合わせるなどしてつけるが，そのルールはプログラマが考え，管理する．操作関数を作る手間を惜しんで，いちいち

```
vec0.x = vec1.x + vec2.x;
vec0.y = vec1.y + vec2.y;
```

などと書いていれば，実用的なプログラムでは，加算は何度も行うはずなのでかえって手間になるし，なにより何をしたいのかが一見してわからない．コピー&ペースト後の変数名の変更し忘れがおき，不幸になることが確実である．

2.5 抽象データ型は関数の作り方の指針

関数の宣言方法はわかっていても，なにを関数にすればよいのかわからなかったかもしれない．抽象データ型の考えにもとづけば，**あるデータ型をあつかう関数は，その型のデータの操作として意味のある関数にする**という，関数の作成指針が明確になる．

プログラムを作る際は，定義されている小さな型から，大きな型を作っていく．関数も，データに必要な操作ごとに，小さな型のための関数を組み合わせて大きな型のための関数を作ってゆく．こうすることで抽象度の高いレベルで処理を簡潔に記述でき，自然ときれいな —— つまり読みやすくミスの起きにくい —— プログラムが書ける．

2.6 演習課題

演習 1. C 言語による 2 次元ベクトル型のソースファイルが vector2c.h および vector2c.c として，またそれを使った簡単なプログラムが vec2ex1c.c として，それぞれ用意されている．以下を行い，確認せよ．

(1) プログラムのコンパイルし，動作させよ．

(2) ベクトルの減算の関数を vector2c.h および vector2c.c に作成追加し，また，vec2ex1c.c の主プログラムに減算を追加し，正しく動くことを確かめよ．

ソースファイルの構成法とコンパイル

C や C++ では，原則として，以下のようにソースファイルを分割する．

(a) **ヘッダファイル**
定数，型の定義 (具体的には，構造体 (や C++ のクラス) の定義になる)，関数のプロトタイプ宣言を，名前が.h で終るファイルにまとめる．これを，ヘッダファイルという．ヘッダファイルは，関連する型 (1 つの場合もある) ごとにまとめる．

(b) **プログラムファイル**
C や C++ の関数の処理を記述するファイルをプログラムファイルという．プログラムファイル名は，C 言語では .c で，C++ 言語は拡張子で区別し，.cc, .c++, .cpp などとする．プログラムファイルの先頭では #include で必要なヘッダファイルをインクルードする．

プログラムファイルは，例えば，それぞれの型毎に 1 つずつにわけ，特定のアプリケーションの主プログラムなども，型とは別のファイルにする．このようにすると，例えば上の例では，2 次元ベクトルを用いた主プログラムを変更しても，2 次元ベクトルの加算，減算などの関数のファイルには影響がないことになる．

複数のソースファイルから成るプログラムをコンパイルするには，

```
% cc vector2c.c vec2ex1c.c
% c++ vector2p.cpp vec2ex1p.cpp
```

のように，必要なソースファイル名を並べればよい．また，

```
% cc -c vector2c.c
% cc -c vec2ex1c.c
```

のように，個々のファイル毎にコンパイルしておき，後で，

```
% cc vector2c.o vec2ex1c.o
```

のようにリンクだけ行なうこともできる．これを**分割コンパイル**という．

3　C++言語で型を作る

3.1　C 言語での型作りの弱点

C 言語で抽象データ型を作ることはできるが，ソースコードがもっと複雑で大規模になった場合, 複数のプログラマで分担して作業するような場合に，次のような問題や不都合が生じる．

(1) 構造体のメンバ変数にはどのような状況でも直接アクセスができ変更できるが，意識せずその一部を変更した場合には，意図しないことが起きうる．プログラマの心がけに任せるばかりでなく，むしろ積極的に，たとえば vector2 をベクトルとして利用する部分では

```
vec0.x = vec1.x + 2 * vec2.x;
```

のような，意味がはっきりしないメンバ変数の操作はできないようにしたい．

(2) 加算には，2 次元ベクトルの加算だけでなく，3 次元ベクトルや行列の加算もある．それらは「加算」という意味では同じであるが，引数の型が異なるため addv2，addv3，addmatrix のようにそれぞれ別の関数名を付ける必要がある．命名法の管理と利用が煩わしい．

(3) 変数の宣言や動的割り付けを行った直後，その変数の各メンバ変数の値がどうなっているのか，あるいはどうする必要があるのかはプログラムコードからははっきりしない．それぞれの初期値が 0 や NULL なのか，それ以外なのか，あるいは不定なのかは，コンパイラや OS に依存する．メンバ変数に抽象データ型が使用され，入れ子になっている場合，どこまで初期値の面倒をみればいいのかも不明である．

3.2　C++言語ではクラスによって型作りが楽になる

C++ では，こののような問題を解決するために**クラス**というデータ型定義の機構が提供されている．クラス class は，C 言語の構造体 struct を拡張したもので,「データを一まとめにする」機能に加え，初期化や操作，アクセスなどそのデータに必要な関数も一まとめにすることができ，またそのデータの内部，つまりメンバ変数や関数へ外からアクセスできるかどうかの制御もできる [1]．これは**データと手続きのカプセル化 encapsulation** と呼ばれ，オブジェクト指向プログラミングでの考え方の一つである．また，変数の宣言や動的領域の確保など，新たにデータ領域が用意される時，明示しなければデフォルトの初期化関数が自動的に呼び出される．また引数を与えて初期化することもできる．必要な初期化は必ず行われるよう保証できるのである．これも，オブジェクト指向プログラミングでの考え方の一つである．

2 次元ベクトル型をクラスで表現すると，たとえば以下のようになる．

```
class Vector2 {
// メンバ変数
```

[1] C++ では構造体 struct も関数を持てるよう拡張されており，class と struct の違いは，デフォルトが private か public か，というだけである．

```
private: // 隠す
        double x;       // x 値
        double y;       // y 値

// メンバ関数
public: // 隠さない／隠せない
        Vector2(void)   { }
                // デフォルトコンストラクタ (初期値不定)
        Vector2(double x0, double y0);
                // x0, y0 を与えて定数 (x0, y0) を得るコンストラクタ
        Vector2 add(Vector2 u);
                // 自身のベクトルとベクトル u との和を求める
        ....
};
```

このように，メンバ変数と同様に，その型専用の関数も記述できる．

クラスでのメンバ変数の宣言は，構造体での宣言と同じである．ただし，アクセス指定子 `private:` によって，メンバ変数 `x` と `y` をこのクラスのメンバ関数 (後述) だけがアクセスできる「プライベート (私的) な」変数とすることができる．これにより，メンバ変数を意識せず —— つまりクラスの定義以外の場所で —— 変更するプログラムは，コンパイルエラーとなる!

データのクラスにそなわるメンバ関数を呼び出すときは，構造体のメンバ変数を参照するときと同様に，変数に `.` で結び付けた形で記述する．例えば，

```
Vector2 v1, v2, v3;
...
v3 = v1.add(v2);
```

という形で呼び出す．これは，C 言語の場合には，

```
v3 = addv2(v1, v2);
```

のように書いていたプログラムに相当する．クラス型に備えられた関数は，その第一引数は，つねにクラス型データが第一引数となる．

クラス内で宣言したメンバ関数の実体は，例えば次のようにプログラムする．

```
Vector2 Vector2::add(Vector2 u)
{
        Vector2 ret;

        ret.x = x + u.x;
        ret.y = y + u.y;

        return ret;
}
```

ここで，ただの `x` と `y` は `add` 関数を呼び出した (作用する) 変数のメンバ変数，つまり `v1.add(v2)` では `v1` のメンバ変数 `v1.x`, `v1.y` を指す．

上の例では，メンバ関数はアクセス指定子 public: によって外部から使うことができる「公開された」関数になっている．メンバ変数，メンバ関数は，それぞれ private: (そのクラス以外からはアクセス不可能)，あるいは public: (そのクラス以外からもアクセス可能) にできる．メンバ変数を private: に，メンバ関数を public: にするのが典型的である．

3.3 C++言語では変数の初期化を明示的に記述する

クラス名と同じ関数名のメンバ関数は，**コンストラクタ**と呼ぶ．コンストラクタはそのクラス（型）の変数の場所をメモリ上に確保するときに呼び出され，その変数が持つメンバ変数の値を設定する．その返り値はそのクラス型のデータ (オブジェクト) と決まっており，型は記述しない．

変数の初期化し忘れを防ぐために，コンストラクタは必ず呼び出されることになっている．コンストラクタは引数を持つこともでき，初期値を引数で指定したり，他の型のデータを初期化に使うものも作成できる．引数の無いコンストラクタは初期値を与えない変数宣言などで暗黙のうちに呼び出されるので，特に **デフォルトコンストラクタ** と呼ばれる．

コンストラクタは，変数宣言などでプログラマが意識して呼び出す場合のほかに，代入 = や型変換 (type cast) で気づかないうちに呼び出される場合もあり，プログラムのミスの原因になることがある [2]．代入やコピーで呼び出されるコンストラクタは**コピーコンストラクタ**とよばれ，引数は同じクラスのデータへの参照 & 一つのみである．C++ プログラミングでのミスを減らすには，自分で作るクラスにはデフォルトコンストラクタ，コピーコンストラクタはとりあえず作っておく習慣をつけるとよい．

この & 演算子は，C 言語ではポインタが指すアドレス（ポインタ型変数のもつ値）を変数から作るために使うが，C++ 言語では，さらに関数の引数で「型名 & 変数名」のように使うことができる．これは，ポインタ型を介さずに，コピーではなく元の変数を参照するという意味である．

演習 2. (C++言語による) 2次元ベクトル型のソースファイルが，vector2.h および vector2.cpp として，またそれを使った簡単な主プログラムが vec2ex2.cpp として，それぞれ用意されている．以下の演習を実施せよ．

(1) プログラムをコンパイルし，実際に動くことを確かめよ．

(2) クラス Vector2 の減算のメンバ関数を vector2p.h および vector2.cpp に宣言追加し，また，vec2ex2.cpp の主プログラムに減算のテストを追加し，減算が行われることを確かめよ．

発展課題 1. C++ 言語では，add など二項演算を行う関数を，自然な二項演算子 + で使えるようにプログラミングできる．

興味があれば，どのようにすればそのように定義できるか，また他にどのような二項演算子をプログラマが定義できるか，調べてみよ．

[2] Smalltalk (Squeak) や Java などの言語ではクラス型データはポインタなど間接参照なのであまり目立たないが，C/C++ ではもともと構造体の代入や関数の引数で値参照が可能なので，注意が必要である．

演習 3. コンストラクタが明示的に，また暗黙の内に選択的に呼び出されることを以下の方法で確かめよ．

(1) デフォルトコンストラクタ `Vector2(void)` で，`x` と `y` に初期値 `0` がセットされるよう変更し，`printf` 等で変数宣言などの直後に初期化されているかどうかを確かめよ．

(2) デフォルトコンストラクタと，コンストラクタ `Vector2(double, double)` の中で `printf` 等でメッセージや値の表示出力を行うようにして，適切にコンストラクタが呼び出されているか確認をせよ．

(3) 値のセットされた `Vector2` 型オブジェクトと全く同じ `x` および `y` 成分の値を持つオブジェクトとして初期化するコピーコンストラクタ `public: Vector2(Vector2 & v)` を宣言し，(2) と同様に呼び出されているかどうかの確認をできるようにして，プログラム中で `Vector2` 型変数への代入を行い確かめよ．

演習 4. 2 次元ベクトル型を参考にして，2 次正方行列型をクラスとして実現せよ．また，簡単な主プログラムを作成し，意図したとおりに動作することを確認せよ．

2 次正方行列型は，少なくとも，入力，出力，加算，減算，乗算の各メンバ関数を持ち，それぞれの動作が確認されること．なお，クラスの実現には，たとえば 2×2 の 2 次元配列を用いればよい．

3.4 C++では関数の引数の型や数が関数名の一部である

コンストラクタは，すべてクラス名と同じである．コンストラクタ以外にも，C++言語では，対象となるクラス（型）や引数，またその数が異なれば，異なる関数に同じ名前を付けることができる．この機能を正しく使うと，例えば次元の異なるベクトルについてそれぞれ加算が定義されているように，扱うデータ型は異なるが人にとって同じ意味や機能を持つ関数を，同じ名前にすることができ便利である．これはオブジェクト指向プログラミング言語に広く取り入れられている考え方の一つで，**ポリモルフィズム** と呼ばれている．

この機能を使うと，まったく異なる意味や機能の関数を同じ名前にすることもできる．つまり，混乱やミスを誘発するような使い方もできるので，注意が必要である．

演習 5. 先に作ったプログラムをあわせて，2 次元ベクトル型と 2 次正方行列型の双方を使うプログラムを作れ．このとき，加算や減算の関数が，同じ名前でもコンパイルでき，正しく動作することを確かめよ．

4 クラスでデータの集合を作る

集合は数学の基本概念であると同時に，その実現方法はプログラミングの重要な基礎である．ここでは，トランプカードをつかったゲームの状況を表現するプログラムを例に，クラスによる集合の実現を学ぶ．

4.1 トランプの手札は，トランプカードの集合である

一組のトランプカードを使うトランプゲームをプログラムするとしよう．そのために，まずカード 1 枚を表すデータ型を考える．

トランプのカードには，スペード ♠，ダイヤモンド ♢，ハート ♡，クラブ ♣ の 4 つの “スート (suit)” [3] があり，それぞれのスートにエース (A)，2 から 10 までの字札，そして Jack (J), Queen (Q)，King (K) のコートカード (絵札) の計 13 枚がある．ここでは，さらに 1 枚ジョーカー (道化師) Joker を加え，計 53 枚の一セット（デッキ）とする．

ここでは，カードのスートと番号を別々のメンバ変数として持つトランプカード型 Card を考える．

```
class Card {
// メンバ変数
private: // カードのスートや番号を後で勝手に変えられても困る
        int suit;       // スート
        int number;     // 番号
// メンバ関数
public:
        ...
};
```

なお，suit の値とスペード，ダイアモンド等との具体的な対応は，enum （列挙型）によって連番でつける．

次に，一人のプレーヤの「手」の表現方法を考える．これは，トランプカードの集合であると考えられる．手札の数は高々数十枚であるので，ここでは集合の実現に配列を用いることにする．

```
class CardSet {
// メンバ変数
// 定義・宣言
public:
        const int maxnumcard = 53;      // 全部持ったとしてもこの枚数
private:
        int numcard;            // 現在の集合内のカード数
        Card cdat[maxnumcard];  // カードの配列
// メンバ関数
...
};
```

カードのデータは配列 cdat に入る．また，実際にカードのデータが納められているのは配列 cdat の添字が numcard 未満の要素であるとする．ここでは，トランプを 1 デッキしか使わないことを仮定しているので，配列は長さ 53 あれば十分である．この 53 という決まった数をプログラムで参照するのに，C 言語では#define コンパイラ指令による書き換えを利用して名前をつけていた．C++ では，const 指定をして，初期化後に値の変更をしない変数，すなわち定数として宣言することができる．

[3] タロットではそれぞれ剣，コイン (またはペンタクル)，カップ，ワンド (木杖) にあたり，アリストテレスの四元素説の 風，土，水，火 を表す．

演習 6. トランプカード型およびトランプカードの集合型のソースファイルが card.[h/cpp] および cardset.[h/cpp] として，またそれを使った簡単な主プログラムが cardsetex1.cpp として，それぞれ用意されている．以下の演習を実施せよ．

(1) プログラムをコンパイルし，動作を確かめよ．

(2) クラス CardSet のメンバ関数 remove() 2 種類を実装して cardset.cpp を完成させ，また cardsetex1.cpp の主プログラムでカード削除の動作テストが行えるようにし，確認せよ．

挿入の手順と削除の手順の共通点に注意し，insert 関数を参考に実装せよ．

5 クラスをもとにより大きなクラスを作る

次に，具体的なトランプゲームをもとに，ゲームの状態を表すクラスを考える．ここでは，最も簡単なカードゲームとして，いわゆる「ババ抜き」[4] を取り上げる．ババ抜きでは，参加プレーヤの持ち手 (持ち札) がゲームの状態を表すと考えることができる．カードの集合型は既に実現しているので，それを利用することを考えると，ババ抜きゲームの状態を表す型は以下のようになる．

```
class BabaState {
// 定義・宣言
public:
        const int numplayer = 5;        // プレーヤ数
// メンバ変数
private:
        CardSet hand[numplayer];  // 各プレーヤの持ち手
// メンバ関数
        ...
};
```

ゲームの状態は，最初に全員にカードが配られた後は，あるプレーヤが別のプレーヤのカードを取り，(プレーヤーが必要とするなら）番号が一致する 2 枚のカードを組で捨てることで進んでいく．カードが配られるときの動作および配られた直後の動作はコンストラクタまたは関数 reset() で，別のプレーヤーからカードを取り，一致する場合捨てる動作を関数 move() で実現しよう．

演習 7. ババ抜きの状態を表現するための型がソースファイル babastate.h および babastate.cpp に，またそれを使った簡単なババ抜きプログラムが babanuki.cpp として，それぞれ用意されている．以下の演習を実施せよ．

(1) BabaState::move() は実現されていない．これを実現し，またゲームを行って動作を確かめよ．

[4] もとは ♣ の Queen (Argine) を抜いて行う Old maid というゲームで，Joker は日本でトランプに追加されたものだという．

6　習うより慣れよ

以下の課題から 1 つまたはそれ以上を選んで取り組め.

演習 8. ここまで，クラス Card は suit と number という二つの整数をメンバ変数に用いて実現してきが，他の方法，例えば一つの char 型で表現する方法も考えることができる.

CardSet クラスやメインプログラムに影響を与えることなく，Card クラスの実現方法を変更できることを，実際に異なる方法で実現し，コンパイル，実行をして確認せよ.

演習 9. ババ抜きの状態を表すクラス BabaState を別の実現方法で実現せよ．例えば，長さ 53 の配列にそれぞれのカードの持ち主を記録しておく方法でも状態を表すことはできる.

新たに実現した方法を比較し，長所と短所，そして結論としてどちらが選りすぐれているかを根拠をしめしながら論ぜよ.

付録： レポートとその提出方法

以下の標準的なレポートの形式に準じているか，よく確かめること．いうまでもないが，ワープロソフトウェアを使う場合も，書式等に自分で責任を持つこと．

1. A4, 縦置き，横書き，左上をステープラどめ．余白は上下 25〜35mm，左右 22 〜 30mm
2. ワープロソフトなどを使う場合，文字スタイルは，本文文章は明朝体，タイトルや節見出しはゴシック．英数文字は半角．プログラムリストやコンソール出力は等幅（非プロポーショナル）体（Courier, Lucida, Consoals など）を使う．ページ番号は，フッタ中央またはヘッダ右端に通しでつける．
3. 文章の部分の行数，行字数，文字サイズは 30 行/ページ 程度（行間 1.25 〜 1.5 行），40 〜 45 文字/一行 程度（文字大きさ 11 〜 12pt）
4. 表題は 18pt 程度で，表紙または一ページ目に本演習のテーマ名，学生番号，氏名，実験日（３回分），提出日を記載する．

内容は一般的なレポートの書き方にそって書くこと．ただし以下が明確にわかるよう書くこと．

1. 演習ごとに，行うこと，あるいは目的の説明.
2. プログラムの自分で作成した部分のプログラムリスト．プログラムのファイル全体は別に電子提出とするので，ファイル名と行番号で参照したり，最初に与えられたファイルそのままの部分，直前の演習と重複する部分などは省略するなど簡潔になるよう工夫すること．
3. 目的の関数などを実現するための考え方，特に自分で考えたこと．
4. 動作の確認においては，確認する個々のことがらと，設計した確認方法，その結果から得られる結論.
5. 考察，この演習で明らかになったこと，考えられる発展的な課題など，のまとめ.

「むつかしかった」など，感想はいらないので，書かない[5] こと．

[5]感想を書きたい場合は，別添やアンケート等でお願いします.