情報工学実験１
オブジェクト指向プログラミング

担当教員名:赤嶺有平
氏名:澤岨千明
学籍番号:0757230G

実験日:2008/07/18
提出日 2008/08/17

課題 1)

なぜ OOP を利用するのか、何故必要とされたのか歴史的な背景を交えて考察せよ。

その考え方が生まれた背景には、計算機の性能向上によって従来より大規模なソフトウェアが書かれるようになってきたということが挙げられる。大規模なソフトウェアが書かれコードも複雑化してゆくにつれてソフトウェア開発コストが上昇した。そこで、ソフトウェアの再利用、部品化といったようなことを意識した仕組みの開発や、ソフトウェア開発工程の体系化などが行われるようになった。

この流れの中、構造化プログラミングができた。そこでデータを構造化し、ブラックボックス化するために考え出されたのが、データ形式の定義とそれを処理する手続きや関数をまとめて一個の構成単とするという考え方がモジュールと呼ばれる概念である。しかし、データはその形式の定義に対して値となる実体（インスタンスと呼ばれる）が複数存在するので、これらをうまく管理する枠組みも必要であると考えた。

そして単なるモジュールではなく、それらのインスタンスを整理して管理する仕組み（例えばクラスとその継承など）まで考慮して生まれたのがオブジェクトという概念である。

課題 2)

「カプセル化」「ポリモーフィズム」「継承」の概念を比喩を用いて説明せよ。

クラスを携帯に例えて表現する。
携帯はボタンを押すとそれに対応した動作をするが、普通に使う分にはその中の仕組みを知る必要はない。勝手に触れると逆に壊れ易くなり、困ったことになるので触れない様にしておく【カプセル化】

携帯の操作方法は同じだが、使う機種によってその性能は異なる。(近年の携帯は昔のそれと比べると容量が大きくなっている) 【ポロモーフィズム】

元々、携帯は一連絡手段として普及していたが、最近では音楽を聞いたり、動画を見たりすることができるようになっている【継承】

課題 3.2)

下記コードにおけるコンストラクタとデストラクタの実行タイミングを printf 文に適宜追加し、その出力から考察せよ。ローカル変数のスコープに注意する。

Number3-2.cpp のソース

```
#include <stdio.h>
class Number
{
 public:
  Number();  // コンストラクタ
  ~Number(); // デストラクタ

  void setNumber(float n);
  float getNumber();
```

```
 private:
  float num;
};

Number::Number()
{
  printf("Construct an instance of the Number\n");
}

Number::~Number()
{
  printf("Destruct the instance of the Number\n");
}

void Number::setNumber(float n)
{
  num = n;
}

float Number::getNumber()
{
  return num;
}

int main(void)
{
// どのタイミングでコンストラクタとデストラクタが実行するか調べる為のもの。
// とりあえず、すべての関数前後に a  b c d … と打ってみました。
  printf("a\n");
  printf("The main function is started.\n");
  printf("b\n");
  Number n;
  printf("c\n");
  n.setNumber(5);

  printf("d\n");
  printf("number is %f\n", n.getNumber() );
  printf("e\n");
  printf("The main function is finished.\n");

  printf("g\n");
  return 0;
}
```

実行結果

```
% ./Number3-2
a
The main function is started.
b
Construct an instance of the Number
c
d
number is 5.000000
e
The main function is finished.
g
Destruct the instance of the Number
```

考察

提示されたソースの int main(void) 内動作の前後に分かり易くするための記述を加えた。
実行結果からもわかるように、b と c の間に
「 Construct an instance of the Number "」 と出ているが、実際のソースでは

```
printf("b\n");
Number n;
printf("c\n");
```

となっている。これは、「Number n;」でコントラクタが呼び出され、「Number::Number()」内の「printf」が実行されたからである。最後に「Dstrruct the instance of the Number」と出力されているのがわかる。実際のソースでは出力が g になっている。このことから、デストラクタが呼びだされるのは main 関数の後であることが考察される。

課題 3.3)

下記コードのコンストラクタとデストラクタの実行タイミングを printf の出力に基づいて考察せよ。特にスタックとヒープの両インスタンスの生成、破棄タイミングの違いに注意する。

Number3-3.cpp のソース

```
#include <stdio.h>
class Number
{
 public:
  Number();  // コンストラクタ
  ~Number(); // デストラクタ

  void setNumber(float n);
  float getNumber();
```

```
 private:
  float num;
};

Number::Number()
{
  printf("Construct an instance of the Number\n");
}

Number::~Number()
{
  printf("Destruct the instance of the Number\n");
}

void Number::setNumber(float n)
{
  num = n;
}

float Number::getNumber()
{
  return num;
}

int main(void)
{
  printf("a\n");
  printf("The main function is started\n");

  printf("b\n");
  Number* pnum = new Number; //ヒープにインスタンスを生成
  printf("c\n");
  Number num2;               //スタックにインスタンスを生成

  printf("d\n");
  pnum->setNumber(5);
  printf("e\n");
  num2.setNumber(6);
  printf("f\n");
  printf("Number = %f Number2 = %f\n", pnum->getNumber() , num2.getNumber() );

  printf("g\n");
  delete pnum;

  printf("h\n");
```

```
  printf("The main function is finished\n");
  printf("i\n");
  return 0;
}
```

実行結果

```
% ./Number3-3
a
The main function is started
b
Construct an instance of the Number
c
Construct an instance of the Number
d
e
f
Number = 5.000000 Number2 = 6.000000
g
Destruct the instance of the Number
h
The main function is finished
i
Destruct the instance of the Number
```

考察

課題 3.2 のソースと同じように出力に英文字をを加えた。main 関数以外は課題 3.2 ソースと同じものである。「`Number* pnum = new Number;`」でコントラクタを呼び出したことでヒープのインスタンスを生成している。「`Number num2;`」ではスタックのインスタンスが生成された。「`deleat pnum;`」でヒープが消され、デストラクタが呼び出されている。スタックは main 関数が終了した後に呼び出されている。

課題 3.4)

なぜカプセル化する必要があるのか、File クラスを例に説明せよ。

File クラスを定義したものを以下に示す。

```
class File { //File クラスを定義する
public:      //public: 以下のメソッドやメンバ変数はクラスの外からアクセス可能
    File();
    void open(char* filename);
    void write(char* data);
    void read(char* out_data);
```

```
    void close();

    FILE* file_descriptor;
}
```

この File クラスのメンバ変数 `file_descriptor` は、クラスの外から書き換えられるべきではない。ファイルディスクリプタというのはプログラムがアクセスするファイルや標準入出力などを OS が識別するために用いられる識別子のことである。OS はこの識別子によってどのファイルを操作するのか判断する。

private を用いてカプセル化することで、クラス外からのアクセスを禁止にできる。

課題 3.5)

下記コードを用いて、スーパークラス (Number) 及びサブクラス (ComplexNumber) のコンストラクタの呼び出しタイミングを調べよ (Number クラスの宣言・定義を追加する必要がある)。

Number3-5.cpp のソース

```
#include <stdio.h>
class Number // Number クラス
{
public:
  Number();  //コンストラクタ
  ~Number(); //デストラクタ

  void setNumber(float n);
  float getNumber();

private:
  float num;
};

Number::Number()
{
  printf("Construct an instance of the Number\n");
}

Number::~Number()
{
  printf("Destruct the instance of the Number\n");
}

void Number::setNumber(float n)
{
  num = n;
}
```

```
float Number::getNumber()
{
  return num;
}

class ComplexNumber : public Number // Number クラスを継承して複素数クラスを定義する
{
public:
  ComplexNumber();
  ~ComplexNumber();
  void setImaginary(float i); // 虚数部を設定
  float getImaginary();       // 虚数部を読み出す

private:
  float imaginaryNumber;
};

ComplexNumber::ComplexNumber()
{
  printf("Construct an instance of ComplexNumber\n");
}

ComplexNumber::~ComplexNumber()
{
  printf("Destruct the instance of ComplexNumber\n");
}

// set(get)Imaginary の定義は各自で記述

void ComplexNumber::setImaginary(float i)
{
  imaginaryNumber = i;
}

float ComplexNumber::getImaginary()
{
  return imaginaryNumber;
}

int main(void)
{
  printf("a\n");
  ComplexNumber cn;
  printf("b\n");
  cn.setNumber(5);
```

```
  printf("c\n");
  cn.setImaginary(0.5);

  printf("d\n");
  printf("ComplexNumber = (%f,%f)\n", cn.getNumber(), cn.getImaginary() );
  printf("e\n");
  return 0;
}
```

実行結果

```
% ./Number3-5
a
Construct an instance of the Number
Construct an instance of ComplexNumber
b
c
d
ComplexNumber = (5.000000,0.500000)
e
Destruct the instance of ComplexNumber
Destruct the instance of the Number
```

考察

ComplexNumber クラスで、「ComplexNumber」「~ComplexNumber」、「setImaginary」「getImaginary」は public なのでクラス外、つまりユーザからアクセスできる。「imaginaryNumber」は private なのでカプセル化されている。

main 関数の「ComplexNumber cn;」で Number と ComplexNumber のコンストラクタが同時に呼び出されている。「retrun 0;」では「ComplexNumber」「Number」の順にデストラクタが呼び出されている。

課題 3.6)

main 関数のループは 3 回反復するが、それぞれどのクラスの print メソッドが呼び出されているか調べよ。またその理由を述べよ。

Number3-6.cpp のソース

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

class Printable { //print 可能な抽象クラス
  //Printable は機能を表すクラスである．このクラスは単体では使用できない．
```

```
  //なぜなら，print するべき対象が定まっていないからである．
public:
  Printable() {} //コンストラクタは何もしない

  virtual void print() = 0; // = 0 は純粋仮想関数であることを示す．サブクラスで定義する必要がある
};

class PrintableNumber : public Printable { //print 可能な Number クラス
public:
  PrintableNumber(float n) { num = n; } //コンストラクタに引数を与えると生成時に値を渡すことができる
  //宣言と定義を同時に行うこともできる

  virtual void print();
  float getNumber() { return num; }
private:
  float num;
};

void PrintableNumber::print() {
  printf("number = %f\n", getNumber());
  printf("void PrintableNumber \n");
}

class PrintableString : public Printable { //print 可能な文字列クラス
public:
  PrintableString(char* s) { strcpy(str_buf, s); }

  virtual void print();
  char* getString() { return str_buf; }
private:
  char str_buf[256]; //文字列を保存するバッファ．本来は必要なサイズをヒープに確保するべき
};

void PrintableString::print() {
  printf("string = %s\n", getString() );
  printf("void PrintableString \n");
}

int main(void)
{
  Printable* plist[3] ;

  plist[0] = new PrintableNumber(5); //コンストラクタに引数を指定できる
  plist[1] = new PrintableNumber(6);
```

```
  plist[2] = new PrintableString("some string");

  int i;
  for(i=0; i<3; i++) {
    plist[i]->print(); //同一のメソッド呼び出しに対して異なるルーチンが呼び出される
    //plist[i] のインスタンスのクラスに応じてそれぞれの print メソッドが呼び出される
  }
}
```

実行結果

```
% ./Number3-6
number = 5.000000
void PrintableNumber
number = 6.000000
void PrintableNumber
string = some string
void PrintableString
```

考察

1回目では「`plist[0] = new PrintableNumber(5);`」より、`PrintableNumber` クラスのインスタンスを生成している。つまり呼び出された print メソッドのクラスは `PrintableNumber` である。
2回目では「`plist[1] = new PrintableNumber(6);`」も同様に `PrintableNumber` クラスのインスタンスを生成している。よって呼び出された print メソッドのクラスは `PrintableNumber` である。
3回目では「`plist[2] = new PrintableString("some string");`」より、`PrintableString` クラスのインスタンスを生成している。つまり、呼び出された print メソッドのクラスは `PrintableString` である。

課題 3.7)

前説の継承と本説の継承は、意味が異なる。どう異なるのか説明せよ。これは、java における extends と interface の違いに相当する。

前説の継承はクラスの機能を拡張するための継承である。(extends)
　extends は単一継承。一つのクラスからしかできない。
本説の継承は同名の関数に複数の定義をしている。(interface)
　interface は複数継承。

課題 4.1)

lineart の C 版、C++版共に、インスタンスに対するある作業が抜けている。このプログラムでは、大きな問題にはならないが、大規模なプログラムでは致命的な問題となる。何が抜けているのか、また、なぜ大規模プログラムで致命的な問題となるのか述べよ。

このプログラムはデストラクタの記述が抜けている。これはオブジェクトが破棄される時に呼び出される特殊な関数で、そのオブジェクトのために確保した占有メモリ領域を開放して再利用できるように処理などを行うもの。

これが抜けていると、使用可能なメモリ領域が減っていき、システムの性能が低下したり、不安定になったりするメモリリークが発生したりするので問題である。

課題 4.2)

ラインアートの C++版プログラムを参考に、ボールが画面を動き回るプログラムの C++版を作成せよ。

main.cpp のソース

```
/*
 *  main.cpp
 *  lineart の lineart.c, lineart.cpp に当たる部分
 *
 *
 *  色々いじった。
 *
 */

#include <GLUT/glut.h>
#include <stdlib.h> // for malloc

#include ‘‘balls.h’’

static Animation anim(20);

// 描画ルーチン
void display(void)
{
  glClear(GL_COLOR_BUFFER_BIT);
  anim.draw();

  glFlush();
}

// ユーザが何もしないときに呼び出される
void idle(void)
{
  anim.forward();
  display(); //再描画を要求する
  //    usleep(10000); // 動きが速すぎるときはコメントを外す
}
```

```
void init(void)
{

  glClearColor(0.0, 0.0, 0.0, 1.0); //背景色を指定する．この例では黒．


}

int main(int argc, char *argv[])
{
  glutInit(&argc, argv);
  glutInitDisplayMode(GLUT_RGBA);
  glutCreateWindow(argv[0]);
  glutDisplayFunc(display);
  glutIdleFunc(idle);
  init();
  glutMainLoop();
  return 0;
}
```

---

balls.cpp のソース

```
/*
 *  balls.cpp
 *  lineart の lines.c, lines.cpp に当たる。
 *
 *
 *  結構触った
 *
 */
#include <OpenGL/gl.h>
#include <glut/glut.h>
#include <stdlib.h>

#include "balls.h"

//定数定義
#define BALL_RAD 0.1 //ボールの半径

//0 以上 f 未満の実数乱数を返すヘルパー関数
float frand(float f)
{
    return (float)rand() / RAND_MAX * f;
}
```

```
//BALL を移動する
void Ball::forward(int i,int num_balls, Ball* b)
{
  int j;
  float n;
  pos.add(vel); // pos に vel を加算

  // ウインドウの端で跳ね返る
  if(pos.x < -1 || pos.x > 1)
    vel.x = -vel.x;
  if(pos.y < -1 || pos.y > 1)
    vel.y = -vel.y;

  /* ここで、balls.c では void forwardAnimation にあった
     ものを持ってくる。                                    */

  //衝突判定
  for(j=0; j<num_balls; ++j) {
    Ball* b2 = b+j-i;
    if(i!=j) {

      /* 物体 a と物体 b の距離が半径 (BALL_RAD) の 2 倍以下であれば
           衝突している.
           跳ね返りは，物体 a，b の速度を入れ替えることで実装できる */

      /*BALL b と b2 の距離が BALL_RAD の 2 倍以下か？ */
      if( (b->pos.x-b2->pos.x)*(b->pos.x-b2->pos.x)
          +(b->pos.y-b2->pos.y)*(b->pos.y-b2->pos.y)
          <=(2*BALL_RAD)*(2*BALL_RAD) ){

        // 衝突した時の処理
        // 色を変える
        b->col = Color( frand(1), frand(1), frand(1) );
        b2->col = Color( frand(1), frand(1), frand(1) );

      // 跳ね返り

        n = b->vel.x;
        b->vel.x = b2->vel.x;
        b2->vel.x = n;
        n = b->vel.y;
        b->vel.y = b2->vel.y;
        b2->vel.y = n;

      }
```

```
    }
  }
}

//BALL を描画する
void Ball::draw()
{
        glColor3f(col.r, col.g, col.b); //色を指定する

        glPushMatrix();
        glTranslatef(pos.x, pos.y, 0);
        glutSolidSphere(BALL_RAD, 16, 2); //球体を描画する
        glPopMatrix();
}

//ANIMATION を生成する (num:生成するボール数)
Animation::Animation(int num)
{
  balls = new Ball[num];
  num_balls = num;

  int i, j;
  // Animation オブジェクトを初期化する
  for(i=0; i<numBalls(); ++i) {
    // 各ボールを適当な位置、速度、色に設定
    balls[i] = Ball(Vector2(frand(2.0)-1, frand(2.0)-1)
                    ,Vector2(frand(0.001), frand(0.001))
                    ,Color((i & 4) / 4, (i & 2) / 2, i & 1));
  }
}

//ANIMATION を進める
void Animation::forward()
{
  int i;

  for(i=0; i<num_balls; ++i) {
    Ball* b = balls+i;
    b->forward(i, num_balls, b);
        }
}
//ANIMATION を描画する
void Animation::draw()
{
    int i;
```

```
    for(i=0; i<num_balls; ++i) {
        Ball* b = balls+i;
        b->draw();
        }
}
```

balls.h のソース

```
/*
 *  balls.h
 *
 *
 *
 *  いくつか書き換え。
 *
 */

#include "types.h"

class Ball{
 public:
  Ball(){}
  Ball(Vector2 ipos, Vector2 ivel, Color icol){
    pos = ipos; vel = ivel; col = icol; // 初期位置、速度、色を設定する
  }
  void forward(int i, int num_balls, Ball*b); // ボールの移動
  void draw();    // ボールの描写

 private:
  Vector2 pos; // 位置
  Vector2 vel; // 速度
  Color col;   // 色
}; // ボールを表すクラス


class Animation{
 public:
  Animation(int num);
  void setBall(int i, Ball p); // ボールを設定する?
  int numBalls(){ return num_balls;} // ボール数を返す
  void draw(); // ボールを描画
  void forward(); // 進む、つまり動く?
 private:
  int num_balls; // ボール数
  Ball* balls;   // ボールの配列へのポインタ
}; // アニメーションを表す構造体
```

types.h のソース

```
/*
 *  color.h
 *
 *
 *
 *  色々書き換え。
 *
 */

class Vector2 {
 public:
  Vector2(){} // 引数を指定しない
  Vector2(float ix, float iy) { x = ix; y = iy; }
  void add(Vector2 rhs) { // ベクトルの加算
    x += rhs.x; y += rhs.y;
  }
  float x;
  float y;
};

class Color {
 public:
  Color() {} // 引数を指定しない場合は初期化しない
  Color(float ir, float ig, float ib) { r = ir; g = ig; b = ib; }
  float r;
  float g;
  float b;
};
```

Makefile

```
LOADLIBES = -framework glut -framework OpenGL
CC = g++ -Wall

balls:main.o balls.o

main.o:balls.h types.h
balls.o:balls.h types.h
types.o:types.h

clean:
        rm balls *.o *~
```

実行結果

make 実行後、生成された ball* を起動すると C で書かれたものと同様の動きをした。

参考文献

- オブジェクト指向プログラミング
  http://ja.wikipedia.org/wiki/オブジェクト指向プログラミング
- IT 用語辞典 e-Words
  http://e-words.jp/
- java 入門｜継承
  http://www.nextindex.net/java/inherit.html