オブジェクト指向言語C++ 後編　～クラスをうまく使おう～

筑波大学情報学群情報メディア創成学類 藤澤誠

1　ヘッダファイル

今回はまずヘッダファイルについて説明します．ヘッダファイルは，主にクラスの宣言，関数のプロトタイプなどを書くファイルで，拡張子は*.h,*.hpp です．前半で示したプログラムをファイルに分けて表示してみます．

ヘッダファイル (vec3.h) には，クラスの宣言部を書きます (List.1)．最初と最後にある`#ifdef,#define,#endif`は，このファイル内のクラス，関数が 2 回以上コンパイルされるのを防ぐための処理です (インクルードガード)．List.2 の実装ファイル (vec3.cpp) には，`Vec3` クラスの実装を記述します．ここでは，コンストラクタ，デストラクタ，`input,output` クラスの中身を vec3.cpp に記述します．このクラスを使うときは，呼び出し側の関数があるファイル (例えば，main.cpp など) の先頭で，`#include "vec3.h"`を記述します．”vec3.cpp”はインクルードする必要ありません．ただし，リンク時に cpp ファイルをコンパイルしたオブジェクトファイルをリンクする必要があります (Visual C++は cpp ファイルがプロジェクトに含まれていれば自動でやってくれます)．

List.1(vec3.h)

```
#ifndef _VEC3_H_
#define _VEC3_H_
#include <iostream>
using namespace std;
//クラスの宣言
class Vec3
{
private:
  float x, y, z;
public:
  //コンストラクタとデストラクタ
  Vec3();
  Vec3(float x0, float y0, float z0);
  ~Vec3();
public:
  //入出力関数
  void input(float x0, float y0, float z0);
  void output(void);
};
#endif
```

List.2(vec3.cpp)

```
#include "vec3.h"
//クラスの実装
Vec3::Vec3()
{
  cout << "constructor" << endl;
}
Vec3::Vec3(float x0, float y0, float z0)
{
  x = x0; y = y0; z = z0;
  cout << "constructor2" << endl;
}
Vec3::~Vec3()
{
  cout << "destructor" << endl;
}
void Vec3::input(float x0, float y0, float z0)
{
  x = x0; y = y0; z = z0;
}
void Vec3::output(void)
{
  cout << "v␣=␣(" << x << ",␣";
  cout << y << ",␣";
  cout << z << ")" << endl;
}
```

後半では，このファイルをインクルードして使っていきますが，これだけでは機能が少ないので，いろいろと改良したバージョンを用います．改良バージョンの Vec3 クラスでは，Vec3 型同士の四則演算，double 型との四則演算，内積，外積，ベクトルの正規化などの機能を持たせています．また，配列のように要素に参照できるようにしています (Vec3 v とすると，v[0],v[1],v[2] で (x,y,z) の値を参照できる)．情報メディア実験のページの最初のサンプルファイルにはこれらの機能+2 次元，4 次元ベクトルの機能も追加した，rx_vec.h が含まれていますので，興味があれば参照してください．

2 継承 (Inheritance) とは

OPP の 2 つ目の柱である**継承**は，簡単に言えば，既存のクラスの定義をベースとして新しいクラスを定義することです．継承では，**子クラス** (もしくは**派生クラス**) に主要な機能を継承することによって，**親クラス** (もしくは**基本クラス**) の機能を拡張します．

2.1 Triangle クラス

それでは，実際にクラスの継承を使ってみましょう．せっかくなので，CG で使えるものにしましょう．CG では，サーフェスモデルをポリゴンで表します．ポリゴンの中でもよく使うのが三角形ポリゴンです．これをクラスで表してみましょう．

三角形ポリゴンの属性として以下のものを考えます．

- 頂点 (3 個)：`Vec3 vertex[3]`
- 法線：`Vec3 normal`
- 材質 (拡散色とする)：`Vec3 color`

将来性を考えて，他のプリミティブ (球やボクセルなど) についても扱えるようにしたいとします．この場合，三角形ポリゴンの属性のうち「材質」は他のプリミティブでも共通になります．そのため，プリミティブ全体を包括する基本クラスとして共通属性を持つ `Object` クラスを作り，その派生クラスとして，三角形ポリゴンを表す `Triangle` クラスを作ります．これらのクラスは，クラス宣言部を object.h，実装部を object.cpp に記述し，`main` 文のある main.cpp でインクルードし，使います．

List.3 に object.h，List.4 に object.cpp，List.5 に main.cpp を示します．

まず，基本クラスである `Object` クラスでは，プリミティブの拡散色 `color` を定義し，コンストラクタにおいてその初期化を行っています．派生クラス `Triangle` では，`Object` クラスを継承しています．継承する場合の書式を以下に示します．

```
class 派生クラス名 : 継承修飾子 基本クラス名
{
--------クラスの内容----------
};
```

ここで継承修飾子は以前説明したアクセスコントロールです．クラスの継承では，基本クラスのメンバのうち，自クラス内，派生クラス，外部からの参照の可否を継承修飾子でコントロールします (以前説明したときは内部と外部の 2 つでしたが，今回は派生先のクラスが追加されていることに注意してください)．継承修飾子は `public`，`private` と `protected` のいずれかです．それぞれ指定した場合の挙動を示します．

継承修飾子	基本クラスの変数	派生クラスにおける変数の扱い	派生クラスオブジェクトからのアクセス
public	private	×	×
	protected	protected	×
	public	public	◯
private	private	×	×
	protected	private	×
	public	private	×
protected	private	×	×
	protected	protected	×
	public	protected	×

少し見づらいですが，「派生クラスにおける変数の扱い」欄はそれぞれの基本クラスの変数が派生クラス内においてどのようなアクセスコントロールを持つ変数として扱われるかを示しています．× はアクセスできないことを示します．例えば，`private` 継承したときの基本クラスの `public` 変数は派生クラスでは `private` になるので，派生クラスのオブジェクトからはアクセス不可になります．また，`public` 継承したときの `protected` 変数は外からはアクセスできませんが，派生クラスからさらに派生した孫クラスからはアクセスできることになります．

ここで，プログラムをみてみると変数 color のアクセスコントロールとして，protected を設定しています．そのため，派生クラスからアクセスできます．そして，Triangle クラスに public 継承しているので，派生クラスである Triangle クラスは，変数 color を protected として扱います．

List.3(object.h)

```
#ifndef _OBJECT_H_
#define _OBJECT_H_
#include "vec3.h"
class Object
{
protected:
  Vec3 color;    //拡散色
public:
  //コンストラクタとデストラクタ
  Object();
  ~Object(){};
};

class Triangle : public Object
{
protected:
  Vec3 vertex[3];  //三角形頂点
  Vec3 normal;  //法線
public:
  //コンストラクタとデストラクタ
  Triangle(Vec3, Vec3, Vec3);
  ~Triangle(){};

  //アクセスメソッド
  Vec3* Normal(void){ return &normal; }
  Vec3* Color(void){ return &color; }
  Vec3* Vertex(int i)
  {
    if(i >= 0 && i <= 2){
      return &vertex[i];
    }
    else{
      return NULL;
    }
  }
};

#endif // _OBJECT_H_
```

List.5(object.cpp)

```
#include "object.h"
Object::Object()
{
  color = Vec3(1.0, 0.0, 0.0);
}

Triangle::Triangle(Vec3 v0, Vec3 v1, Vec3 v2)
{
  vertex[0] = v0; vertex[1] = v1; vertex[2] = v2;
  //法線を計算
  normal = (v1-v0).cross(v2-v0);
  normal.normalize();
}
```

List.6(main.cpp)

```
#include "object.h"
int main(void)
{
  Triangle *tri;
  Vec3 v0, v1, v2;
  v0 = Vec3(0.0, 0.0, 0.0);
  v1 = Vec3(1.0, 0.0, 0.0);
  v2 = Vec3(1.0, 1.0, 0.0);

  //オブジェクト生成
  tri = new Triangle(v0, v1, v2);

  //三角形ポリゴン情報の出力
  cout << "color = ";
  tri->Color()->output();
  cout << "normal = ";
  tri->Normal()->output();
  for(int i = 0; i < 3; ++i){
    cout << "vertex " << i << " = ";
    tri->Vertex(i)->output();
  }

  delete tri;
  return 0;
}
```

このプログラムの出力結果は,

```
color = (1, 0, 0)
normal = (0, 0, 1)
vertex 0 = (0, 0, 0)
vertex 1 = (1, 0, 0)
vertex 2 = (1, 1, 0)
```

2.2 継承とコンストラクタ

前節で継承を使ったプログラムをやりました．この中で，ちょっと不思議に思ったことはありませんか?プログラムの出力結果を見てみると，color = (1, 0, 0) となっています．main の中では色は設定していません (というより変数 color は派生クラスのオブジェクトからはアクセスできないようになっています)．これは，Object クラスのコンストラクタで設定されています．main の中では派生クラスである Triangle クラスのオブジェクトを生成しているだけです．つまり，派生クラスのオブジェクトを生成すると，基本クラスのコンストラクタも実行される，ということです．これは，デストラクタでもいえます．これを確かめるために，それぞれのコンストラクタを以下のように書き換えてみます.

```
Object::Object()
{
  cout << "Object::Object running..." << endl;
  color = Vec3(1.0, 0.0, 0.0);
}

Triangle::Triangle(const Vec3 &v0, const Vec3 &v1, const Vec3 &v2)
{
  cout << "Triangle::Triangle running..." << endl;
  vertex[0] = v0; vertex[1] = v1; vertex[2] = v2;
  // 法線を計算
  normal = (v1-v0).cross(v2-v0); normal.normalize();
}
```

実行結果はこうなります.

```
Object::Object running...
Triangle::Triangle running...
color = (1, 0, 0)
  ...
```

このことから，派生クラスのオブジェクトを生成した場合，**基本クラス** → **派生クラス**の順番でコンストラクタが実行されることが分かります．継承では，**多重継承**といって派生クラスからさらに派生させることができます．このときは，基本クラスから順番に実行されていきます.

では，基本クラスのコンストラクタが引数を持つ場合はどうなるのでしょう．以下のように書き換えて実行してみましょう.

```
Object::Object(Vec3 col)
{
  color = col;
}
```

これは，実行するどころかコンパイルでエラーとなると思います．これを解決するためには，明示的に基本クラスのコンストラクタを呼ばなくてはなりません．基本クラスのコンストラクタを明示的に呼ぶには，派生クラスにおいて `Object::Object(col)` と実行すればよいのですが，問題はそれをどこに書くかと言うことです．基本クラスのコンストラクタが実行される前に書く必要があるのですが，先程も述べたとおり，基本クラスのコンストラクタ → 派生クラスのコンストラクタの順で実行されるので，コンストラクタ内に書いたのでは間に合いません．そのため，関数定義の本体ではなく，その前で基本クラスのコンストラクタを呼び出します．

```
Triangle::Triangle(Vec3 v0, Vec3 v1, Vec3 v2, Vec3 col) : Object(col)
{
    .......
}
```

これを**イニシャライザ**と呼びます．

3 コンポジション

List.3 の `Object` クラス，`Triangle` クラスでは，クラスのメンバとして，`Vec3` クラスのオブジェクトを持っています．このように，他のクラスのメンバを持つクラスのことを**コンポジション**といいます．

3.1 is a と has a

コンポジションでは，内部に別のオブジェクトを持つので，2 つのオブジェクトの機能を使うことができます．そういう意味では継承と似ています．

では，継承とコンポジションはどう使い分ければよいのでしょうか? これは，好みによるのですが，普通はオブジェクト間の関係を **is a 関係**と **has a 関係** (もしくは **parts of 関係**) に分けて考えます．

- **is a 関係**
  継承は，他のオブジェクトの機能拡張であり，継承先のオブジェクトは，継承元のすべての機能を含みます．つまり，一方が一方の一種であるという関係です．これを is a 関係といいます．

  例えば，三角形ポリゴン is a 3D オブジェクト，猫 is a 動物などです．

- **has a 関係**
  コンポジションは，クラス型のクラスのオブジェクトを所有する関係です．これを has a 関係と呼びます．

  例えば，三角形ポリゴン has a 頂点 1，車 has a エンジンなどです．

4 多態性

多態性 (ポリモーフィズムとも呼ばれる) とは,「多くの態 (すがた) を持つ」という意味です．これは，同じ名前のメッセージを送っても、受け手によって適切な手続きが呼び出されることです．

前回作成したクラスを例に説明します．図 1 に前回作成した `Object` クラスと `Triangle` クラスを図で示しました．

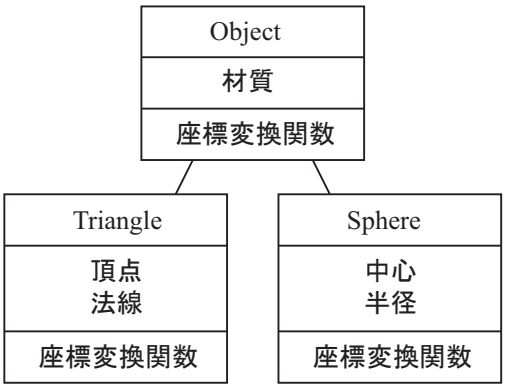

図 1: 多態性

図 1 では，さらに Sphere(球) クラスを Object クラスの派生クラスとして追加しています．各クラスを表す四角形内の各カラムでは，上からクラス名，属性 (メンバ変数)，メソッド (メンバ関数) を示します．Sphere クラスの属性は，球の中心座標と半径としています．

今，このクラスに，各オブジェクトの座標変換関数 (Translate,Scaling,Rotate) を追加する場合を考えます．まず考えられるのは，各クラスごとに別名の関数 (例えば，TranslateTriangle,TranslateSphere) を用意する方法です．この方法では，各クラスのオブジェクトごとに呼び出す関数を変える必要があり，面倒です．一番便利なのは，Object クラスのオブジェクトを用意して，そこから Triangle クラスの座標変換関数，Sphere クラスの座標変換関数にアクセスする方法です．クラスでは，この機能を実現できます．このとき，Object クラスのオブジェクトは，ある時は Triangle として振る舞い，ある時は Sphere として振る舞います．これを多態性と呼びます．

4.1 関数のオーバーライド

前半で関数のオーバーロードというものをやったことを思い出してください．関数のオーバーロードでは，「異なる引数を持つ同じ関数名の関数を複数持つことができる」というものでした．クラスの継承では，関数の**オーバーライド**と呼ばれることができます (名前が似ているので気を付けてください)．オーバーライドでは，派生クラスで基本クラスの関数を書き換えて，処理内容を変更することです．ここでの同じ関数とは，関数名だけでなく，引数も同じと言うことを示しています．List.7 にその例を示します．

List.7 は前回までに作成したプログラムに Output 関数を追加しただけです．これは，各オブジェクトの情報を出力する関数です．次のように実行した場合，

```
Object obj;    // Object クラスのオブジェクト
Triangle tri;  // Triangle クラスのオブジェクト
obj.Output();  // ここでは，Object クラスの Output 関数が実行される
tri.Output();  // ここでは，Triangle クラスの Output 関数が実行される
```

実行結果は，

```
Object Output
....
Triangle Output
....
```

となります．Triangle クラスの Output により Object クラスの Output 関数がオーバーライドされたので，Triangle クラスのオブジェクトを使って Output 関数を呼び出すと，Triangle クラスのものが呼び出されます．

もし，派生クラスから基本クラスのオーバライドされた関数の方を呼び出したい場合は，**スコープ解決演算子**を用います.

```
Triangle tri;           // Triangle クラスのオブジェクト
tri.Object::Output();   // Object クラスの Output 関数が実行される
```

Triangle クラスのメンバ関数内から呼び出す場合もこのスコープ解決演算子を用います.

List.7

```
class Object
{
  ....
public:
  void Output(void)
  {
    cout << "Object␣Output" << endl;
    ....  //属性情報などの出力
  }
};

class Triangle : public Object
{
  ....
public:
  void Output(void)
  {
    cout << "Triangle␣Output" << endl;
    ....  //頂点，法線情報などの出力
  }
};
```

4.2 オブジェクトの代入互換性

さて，以前 new,delete 演算子でオブジェクトを動的に生成する方法を説明しました．このとき，オブジェクトへのポインタを用いました．このオブジェクトへのポインタは，継承に関連した性質として，**オブジェクトの代入互換性**を持ちます．これは，

基本クラスのポインタに派生クラスのポインタを代入できる

という性質です.

Object - Triangle クラスの例を以下に示します.

```
Triangle tri;
Object *obj1 = &tri;                // 静的に生成した Triangle オブジェクトのポインタを代入
Object *obj2 = new Triangle(...);   // new で生成した Triangle オブジェクトのポインタを代入
```

このようにObjectクラスのポインタにTriangleクラスのオブジェクトのポインタを代入できます．しかし，この逆はできません．つまり，派生クラスのポインタに基本クラスのポインタを代入することはできないということです．これについて考えていきます．

まず，オブジェクトの代入互換性を関数オーバーライドと組み合わせて考えます．先ほどのList.7において，Output関数を追加しました．これを実行してみます．

```
Object *obj;               // Objectクラスのポインタ
obj = new Object(...);     // Objectクラスのオブジェクトを生成
obj->Output();             // ---(1)
obj = new Triangle(...);   // Triangleクラスのオブジェクトを生成
obj->Output();             // ---(2)
```

実行結果は，

```
Object Output
....
Object Output
....
```

(1),(2)のどちらも基本クラスのOutputが実行されていることが分かります．これを**基本クラスへの回帰**と呼びます．

基本クラスのポインタに派生クラスのポインタを代入可能なのは，基本クラスが派生クラスのサブセットになっているからです(図2)．派生クラスは基本クラスにメンバを追加したものであり，オブジェクトの代入互換性は，派生クラスで追加されたメンバを切り捨てることで実現しています．

このことから，基本クラス → 派生クラスの代入はできないことが分かります．基本型は派生型で追加されたメンバを持っていません．よって，基本クラスのオブジェクトを派生クラスのオブジェクトに代入できたとしたら，派生クラスで追加されたメンバを呼び出そうとしてもないということになってしまいます．

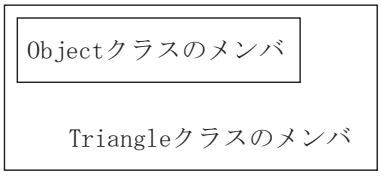

図 2: 基本クラスと派生クラス

さて，ここまできたところで，変だなと思いませんか．ここの説明では，オブジェクトを代入しても結局基本クラスになってしまうのだったら，多態性なんてできないではないかと．

これを解決するのが，次に説明する仮想関数です．

4.3 仮想関数

仮想関数とは，virtualという修飾子を付けて宣言されたメンバ関数です．以下に，List.7のOutput関数を仮想関数化した例を示します．

```
class Object
{
  virtual void Output(void);    // 仮想関数としてvirtualを付ける
};
```

```
class Triangle : public Object
{
  virtual void Output(void);    // virtual は付けても付けなくてもよい
};
void Object::Output(void)       // ここでは virtual を付けない
{
  ....
}
```

派生クラスの Output 関数には，virtual を付ける必要はありませんが，付けておいた方がこれが仮想関数であるということがわかりやすいということもあるので付けることをお勧めします．

さて，仮想関数を使ったらどうなるのでしょう？先ほどと同様に実行してみます．

```
Object *obj;                // Object クラスのポインタ
obj = new Object(...);      // Object クラスのオブジェクトを生成
obj->Output();              // ---(1)
obj = new Triangle(...);    // Triangle クラスのオブジェクトを生成
obj->Output();              // ---(2)
```

実行結果は，

```
Object Output
....
Triangle Output
....
```

となります．(2) で派生クラスの Output 関数が実行されていることが分かります．仮想関数を使った場合，基本クラスへの回帰が回避され，ポインタが派生クラスのオブジェクトを指していれば，派生クラスの関数が実行されます．

このように，オブジェクト obj は，(1) では Object クラス，(2) では Triangle の Output 関数を実行しています．このように，時間とともに姿が変わり，あるときは Object クラス，あるときは Triangle クラス，また，あるときは Sphere クラスとして振る舞うことを多態性と呼びます．

4.4 純粋仮想関数

これまでに作成したクラスに，座標変換関数を追加することを考えます．ここでは，平行移動を行う関数 Translate を考えます．

図 1 に示すように，座標変換関数はすべてのクラスで定義され，また，仮想関数とします．しかし，基本クラスである Object クラスは，3D オブジェクトの属性情報しか持ちません．よって，座標変換はする必要がありません (というよりできません)．一方で多態性を実現するためには，基本クラスにも関数を定義しなくてはならないので，以下のように中身が空っぽの関数を定義してみます．

```
void Object::Translate(const Vec3 &trans)
{
}
```

このように，わざわざ空の関数を定義するのは無駄なことです．こんな時のための機能として，**純粋仮想関数**というものがあります．関数を純粋仮想関数にするときには，関数宣言の最後に = 0 を付けます．純粋仮想関数を使って Translate 関数を追加したものを List.8 に示します．

List.8

```
// クラスの宣言Objectオブジェクト記述用基底クラス –
class Object
{
protected:
  Vec3 color;    //拡散色
public:
  //コンストラクタとデストラクタ
  Object(Vec3);
  virtual ~Object(){}
public:
  //純粋仮想関数
  virtual void Output(void);
  virtual void Translate(const Vec3&) = 0;
};

// クラスの宣言Triangle三角形オブジェクト記述用クラス –
class Triangle : public Object
{
private:
  Vec3 vertex[3];  //三角形頂点
  Vec3 normal;  //法線
public:
  //コンストラクタとデストラクタ
  Triangle(const Vec3&, const Vec3&, const Vec3&, const Vec3& = Vec3(0.0));
  virtual ~Triangle(){}
public:
  //仮想関数
  virtual void Output(void);
  virtual void Translate(const Vec3&);
};

void Triangle::Translate(const Vec3 &trans)
{
        vertex[0] += trans;
        vertex[1] += trans;
        vertex[2] += trans;
}
```

List.8 において，デストラクタも仮想関数となっていることに注意してください．こうしておかないと，以下のような場合に，基本クラスのデストラクタが実行されてしまいます．

```
Object *obj;              // Object クラスのポインタ
obj = new Triangle(...);  // Triangle クラスのオブジェクトを生成
```

```
....
delete obj;
```

この場合，派生クラスである Triangle クラスのデストラクタが実行されるべきです．よって，デストラクタを仮想関数にしておく必要があるのです．

5　フレンドクラスとフレンド関数

前章で多態性について解説しましたが，ちょっとそこから離れて，最後にフレンドクラス，フレンド関数というものを紹介します．

5.1　フレンド関数

フレンド関数とは，クラスのメンバ関数ではなく，クラス外で宣言される一般関数ですが，クラスのメンバ関数と同様にクラスの非公開メンバ (private メンバと protected メンバ) にアクセスできる関数です．

先ほどの，Object クラスにフレンド関数 Input を追加してみましょう．

```
class Object
{
protected:
  Vec3 color;    // 拡散色
public:
  friend void input(Object obj, Vec3 col);
};
void input(Object obj, Vec3 col)
{
  obj.color = col;
}
```

5.2　フレンドクラス

フレンド関数と同じように，クラスの非公開メンバに自由にアクセスできる**フレンドクラス**というものがあります．Object クラスにフレンドクラスとして，Render クラスを追加してみます．

```
class Object
{
protected:
  Vec3 color;    // 拡散色
public:
  friend class Render;
};

class Render
{
protected:
  Object obj;
public:
```

```
  void Output()
  {
    cout << obj.color[0];
    .....
  }
}
```

このように，フレンドクラス，フレンド関数ではクラスの非公開メンバに自由にアクセスできるようになります．しかし，使い方を間違うと，クラスのカプセル化による内部動作の隠蔽を損なうことになります．そのため，便利ではありますが，個人的にはあまり多用するのはお勧めしません．しっかりとクラス設計をしていれば使う必要はないでしょう．