オブジェクト指向と
ゲームプログラミング

○　C++編 － 第７回　ポリモーフィズム　○

ポリモーフィズム

オブジェクト指向の重要な概念のひとつに、ポリモーフィズム(polymorphism：多態性または多相性)があります。ポリモーフィズムは、オブジェクト指向プログラミングで必ず行わなければならないものではありませんが、プログラミングの効率化に効果があります。

ポリモーフィズムは、同じ名前を持った動作が状況に合わせて異なる動作をすることをいいますが、ソースコード上でのそれは、「同じ名前で異なる機能を持った関数を複数定義する」ことをいいます。C++では、関数のオーバーロードまたはオーバーライドにより、これを実現しています。

オーバーロード

グローバル関数またはメンバ関数において、同じ名前の関数を複数定義することができます。これを関数のオーバーロード(overloading：多重定義)と呼びます。オーバーロードする際には、各関数の引数の型または個数が異なるようにしなければなりません。戻り値の型のみが異なる場合は、文法エラーとなり、オーバーロードできません。

```
class CTest {
public:
    void SetValue(const int    val) { n_ = val; }
    void SetValue(const float  val) { f_ = val; }
    void SetValue(const double val) { d_ = val; }

    void SetValue(const int n, const float f, const double d)
    { n_ = n; f_ = f; d_ = d; }

private:
    int      n_;
    float    f_;
    double   d_;
};
```

上のクラスでは、SetValue関数が４つあります。

```
void SetValue(const int    val)
void SetValue(const float  val)
void SetValue(const double val)
void SetValue(const int n, const float f, const double d)
```

このように、同じ名前の関数を同じクラス内に定義することができます。この例では戻り値の型が同じですが、異なっても構いません。オーバーロードされた関数は、引数の型や個数から、それに応じたものが呼び出されます。

```
CTest   test;
test.SetValue(100);             // int型      のSetValue関数が呼び出される
test.SetValue(89.3);            // double型   のSetValue関数が呼び出される
test.SetValue(0, 1.0f, 8.93);   // 引数が３つのSetValue関数が呼び出される
```

オブジェクト指向では、ひとつの動作がその状況に応じて別々の働きを持つことをポリモーフィズムと呼びますが、このオーバーロードも、同じ名前の関数が引数の型や数に合わせた動作をすることからそのひとつだと言えます。

オーバーライド

クラスを継承した場合、基底クラスで定義されたメンバ関数の機能が派生クラスの目的に合わない場合があります。このような場合、派生クラスでそのメンバ関数を再定義し、機能を上書き変更することができます。これを関数のオーバーライド(overriding：再定義)と呼びます。何度も継承を行って階層的な構造になっているクラスのメンバ関数をオーバーライドした場合は、最後にオーバーライドしたものが優先されます。

メンバ関数をオーバーライドするには、「関数名」「引数の型と数」「戻り値の型」のすべてが基底クラスと同じである必要があります(アクセス指定子は変更しても構いません)。
関数名が異なる場合は、当然別の関数として扱われます。引数の型や数が異なるとオーバーロードとなりそうですが、クラスを超えてのオーバーロードは規格上できないので、文法エラーとなります。戻り値の型だけが異なる場合は、オーバーライドもオーバーロードもできず、文法エラーとなります。

```
class Character {
public:
    void Move();
};

class Enemy : public Character {
public:
    void Move();          // Move関数のオーバーライド
};
```

派生クラスでメンバ関数をオーバーライドしても、基底クラスのオーバーライドされた関数がなくなってしまうわけではありません。基底クラスのオーバーライドされたメンバ関数も呼び出すことができるようになっています。
オーバーライドされたメンバ関数を呼び出したい場合は、スコープ解決演算子::を使い「クラス名::関数名」と記述します。たとえば、上のEnemyクラスまたはそのオブジェクトが、CharacterクラスのMove関数を呼び出したい場合は、以下のように記述します。

```
Character::Move();  // CharacterクラスのMove関数を呼び出す
```

基底クラスのオーバーライドする関数に、派生クラス共通の処理を記述しておけば、派生クラスではその差分だけを作成すればよくなり、効率的にプログラムを作成できます。

アップキャストとダウンキャスト

基底クラスのポインタ型変数に、派生クラスのポインタを代入することができます。これをアップキャストと呼びます。たとえば、

```
Character*   chara = new Enemy();   // 派生クラスのポインタを基底クラスのポインタに代入
```

ということができるのです。C++では、ポリモーフィズムや動的オブジェクトを表現するのに、アップキャストが重要な役割を担っています。アップキャストはポインタ以外に、参照型にも行うことができます。なお、ポインタでも参照型でもない変数にアップキャストした場合は、派生クラス独自のメンバがすべて削ぎ落とされ、消滅してしまいます。これをスライシングと呼びます。

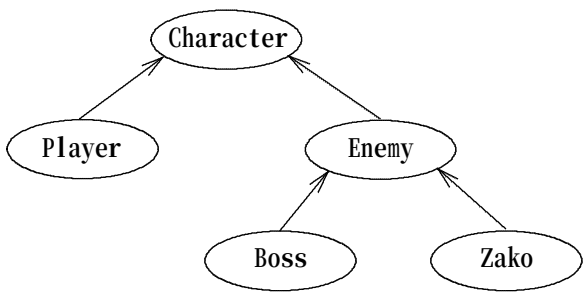
この逆の動作、派生クラスのポインタ型変数に基底クラスのポインタを代入することをダウンキャストと呼びます。しかし、ダウンキャストは禁止されています。なぜかというと、「基底クラスのオブジェクトを派生クラスのオブジェクトとして使う」ことを意味するからです。基底クラスのオブジェクトは、派生クラスのオブジェクトに比べ、メンバの数が足りない可能性があります。もし、Enemyクラスのポインタに Characterクラスのポインタが代入可能だとしたら、Enemyクラス独自のメンバの部分が不定になってしまいます。CharacterクラスにはEnemyクラスで拡張された機能がないからです。

基底クラスのポインタに派生クラスのポインタを代入した場合、派生クラスで追加したメンバは使えなくなります。基底クラスのポインタは、あくまでも基底クラスとみなされるからです。

```
chara->Enemyクラス独自のメンバ;
```

このような参照はできないのです。

アップキャストとダウンキャストという名前の由来は、クラス階層図での方向から来ています。クラス階層図とは、クラスの継承関係を表すもので、次のようなものです。



クラス階層図は、このようなツリーで記述されます。下のクラスは、上のクラスから派生していることを示しています。たとえば、「Player」は「Character」、「Zako」は「Enemy」、「Enemy」は「Character」を継承していることが読みとれます。

アップキャストは、上のクラスに向ってなされ、ダウンキャストは、下のクラスに向ってなされます。このように、アップキャストとダウンキャストは、キャストの方向を示しているのです。

仮想関数

たとえば、以下のようにアップキャストを行ったとします。

```
Character*   chara = new Enemy();
```

こうすると、Characterクラスのポインタが Enemyクラスのオブジェクトを指すことになります。この状態で以下のようにメンバ関数Moveを呼び出してみると、

```
chara->Move();
```

ポインタの指すオブジェクトがEnemyクラスであるにもかかわらず、Characterクラスの Move関数が実行されてしまいます。ところが、Move関数が仮想関数であれば、変数charaが実際に指すクラスのものが呼び出されるようになります。

メンバ関数を仮想関数にするには、基底クラスの関数の宣言でキーワードvirtualを記述します。たとえば、CharacterクラスのMove関数を仮想関数にする場合は、以下のようになります。

```
class Character {
public:
    virtual void Move();        // Move関数を仮想関数にする
};

class Enemy : public Character {
public:
    virtual void Move();        // Move関数のオーバーライド
};
```

キーワードvirtualは、基底クラスのクラス定義(ヘッダファイル)には必要ですが、クラス外部の関数定義では必要ありません。また、派生クラスの対応する関数はvirtualがなくても自動的に仮想関数として扱われます。しかし、仮想関数であることを明示するため、派生クラスのクラス定義でもvirtualをつけるようにします。

オブジェクト指向では、ひとつの動作がその状況に応じて別々の働きを持つことをポリモーフィズムと呼びますが、仮想関数を用いた関数のオーバーライドも、同じ名前の関数がオブジェクトの型に合わせた動作をすることからそのひとつだと言えます。

仮想デストラクタ

基底クラスのポインタ型変数が派生クラスのオブジェクトを指している場合、仮想関数でない関数の呼び出しは、たとえオーバーライドされていたとしても、基底クラスのものが呼び出されます。
new演算子で派生クラスのオブジェクトを動的に生成し、返されたポインタを基底クラスのポインタ型変数に代入した場合、このオブジェクトの解放はdelete演算子で行います。しかし、派生クラスにデストラクタが定義されていても、前述の理由から、基底クラスのデストラクタが仮想関数でなければ、基底クラスのデストラクタしか実行されません。これは、派生クラスが正しく解放されないといった、非常に深刻なバグの原因になります。継承するしないにかかわらず、基底クラスのデストラクタはつねにvirtual指定し、仮想デストラクタにしておきましょう。

```
class Character {
public:
    virtual ~Character();       // 仮想デストラクタにする
    virtual void Move();
};

class Enemy : public Character {
public:
    virtual ~Enemy();           // デストラクタ
    virtual void Move();
};
```

仮想関数テーブル

以下のようなクラスが定義されている場合、基底クラスCharacterには３つの仮想関数があります。

```
// キャラクタークラス
class Character {
public:
    Character();
    virtual ~Character();   // 仮想デストラクタ
    virtual void Move();    // 仮想関数Move
    virtual void Draw();    // 仮想関数Draw
};

// プレイヤークラス
class Player : public Character {
public:
    Player();
    virtual ~Player();
    virtual void Move();
    virtual void Draw();
};

// エネミークラス
class Enemy : public Character {
public:
    Enemy();
    virtual ~Enemy();
};
```

仮想関数を正しく処理するため、仮想関数テーブルというものが作成されます。仮想関数テーブルには、仮想関数ごとにどの実体関数を呼んだらよいのかという情報が格納されます。派生クラスが定義されると、派生クラスにひとつずつ仮想関数テーブルを作っていきます。

コンストラクタのアドレス
デストラクタ実行関数のアドレス
Move実体関数のアドレス
Draw実体関数のアドレス

▲仮想関数テーブルのフォーマット

Character::Character
Character::~Character
Character::Move
Chracter::Draw

▲Characterクラスの仮想関数テーブル

Player::Player
Player::~Player
Player::Move
Player::Draw

▲Plyerクラスの仮想関数テーブル

Enemy::Enemy
Enemy::~Enemy
Character::Move
Character::Draw

▲Enemyクラスの仮想関数テーブル

基底クラスには、仮想関数テーブルを格納する隠しメンバVtblAdrが追加されます。この隠しメンバは、コンストラクタが暗黙のうちに初期化します。基底クラスのポインタを通じて仮想関数が呼ばれたとき、まずこのVtblAdrが指し示す仮想関数テーブルの内容を調べ、そこに登録された実体関数を呼びます。

純粋仮想関数

たとえば、キャラクターの基本的な動作と属性を管理するCharacterクラスを基底クラスに、派生クラスとしてプレイヤーを表すPlayerクラス、ザコ敵を表すZakoクラス、ボス敵を表すBossクラスを作成し、各クラスには移動処理を行うMove関数があるとします。

ポリモーフィズムを実現するため、基底クラスのMove関数は仮想関数とします。しかし、プレイヤーはキーボードによって移動、ザコは決められたルートを移動、ボスはプレイヤーや自身の状態を判断して移動パターンを変えたり、あらかじめ決められたいくつかのパターンをとるなどのように、移動処理は各クラスによって大きく異なります。

基底クラスCharacterでは、各派生クラスで共通する移動処理というものを記述することはできません。これは、Characterクラスでは、Move関数本体の実装ができないということになります。この関数は、派生クラスで実装されるべきなのです。
実装できないので「何も処理をしない」という関数にすることもできますが、このような場合、C++では、関数を純粋仮想関数というものにすることができます。純粋仮想関数とは、仮想関数であるものの実体がない－実装は派生クラスでする、というものです。作り方はとても簡単で、クラス定義内で仮想関数の後ろに「=0」をつけるだけです。こうすると、その関数は純粋仮想関数になります。たとえば、CharacterクラスのMove関数を純粋仮想関数にするには、以下のようにします。

```
class Character {
public:
    Character();
    virtual ~Character();
    virtual void Move() = 0;        // Move関数を純粋仮想関数にする
    virtual void Draw();
};
```

純粋仮想関数は、宣言のみ可能で関数本体の定義を行うことはできません。また、基底クラスの純粋仮想関数として宣言された関数は実体が存在しないので、どのような手段を使っても呼び出すことはできません。

抽象クラス

純粋仮想関数を１つでも持つクラスを抽象クラスといいます。抽象クラスにも、メンバ関数やメンバ変数、仮想関数を定義することができます。
抽象クラスは純粋仮想関数を必ず持っている、つまり関数本体の定義がないメンバがあるため、インスタンスを生成することはできません(抽象クラスのポインタを格納する変数は生成できます)。
抽象クラスは、継承によって派生され、そこで関数の実体が定義されることを前提にしたクラスを作成したり、ポリモーフィズムのためのインタフェイスを提供したりする場合に使用します。

練習問題

1　以下の文章の内容が「カプセル化」の場合はA，「オーバーロード」の場合はB，「オーバーライド」の場合はCを付けましょう。

(1)基底クラスのポインタから派生クラスのメンバ関数を呼び出す。
(2)メンバ変数に不適切な値が書き込まれるのを防ぐ。
(3)クラス内部の変更が、クラスを使う側に影響を与えないようにする。
(4)基底クラスのメンバ関数の処理内容を派生クラスで上書き変更する。
(5)ひとつのクラスに複数のコンストラクタを定義する。
(6)ひとつのクラスに同じ名前の関数を複数定義する。

2　以下の文章の内容が正しい場合には○、間違っている場合には×を付けましょう。

(1)純粋仮想関数とは、関数の処理内容だけを記述した仮想関数である。
(2)純粋仮想関数の定義では、引数や戻り値を指定しない。
(3)純粋仮想関数を1つ以上持つクラスを抽象クラスと呼ぶ。
(4)純粋仮想関数は、それを継承したクラスで必ずオーバーライドしなければならない。
(5)純粋仮想関数は、それを継承したクラスで必ずオーバーロードしなければならない。

3　継承を行ったとき、デストラクタを仮想関数にしなかった場合の問題点について述べましょう。

4　以下のプログラムを実行し、仮想関数の働きを確認しましょう。

```
#include <iostream>

using namespace std;

// キャラクタークラス
class Character {
public:
    virtual ~Character() {}
    virtual void Move() { cout << "キャラクター移動" << endl; }
    virtual void Draw() { cout << "キャラクター描画" << endl; }
};

// プレイヤークラス
class Player : public Character {
public:
    virtual ~Player() {}
    virtual void Move() { cout << "プレイヤー移動" << endl; }
    virtual void Draw() { cout << "プレイヤー描画" << endl; }
};

// 敵クラス
class Enemy : public Character {
public:
    virtual ~Enemy() {}
    virtual void Move()
    {
        cout << "敵移動：";
        Character::Move();
    }
};
```

```
// ボスクラス
class Boss : public Enemy {
public:
    virtual ~Boss() {}
    virtual void Move() { cout << "ボス移動" << endl; }
    virtual void Draw() { cout << "ボス描画" << endl; }
};

int main()
{
    Character*   chara[8];

    // キャラクター生成
    chara[0] = new Player;
    chara[1] = new Enemy;
    chara[2] = new Enemy;
    chara[3] = new Enemy;
    chara[4] = new Enemy;
    chara[5] = new Enemy;
    chara[6] = new Enemy;
    chara[7] = new Boss;

    int   i;

    // 移動
    cout << "--- 移動処理開始 ---" << endl;
    for(i = 0; i < 8; i++)
        chara[i]->Move();

    cout << endl;

    // 描画
    cout << "*** 描画処理開始 ***" << endl;
    for(i = 0; i < 8; i++)
        chara[i]->Draw();

    // キャラクター解放
    for(i = 0; i < 8; i++) {
        delete chara[i];
        chara[i] = NULL;
    }

    return 0;
}
```