オブジェクト指向と
ゲームプログラミング

○　C++編 － 第６回　継承　○

継承

これまでのCharacterクラスは、キャラクターの基本的な機能をまとめたものでした。さらに新しいクラスを作成する場合、たとえば自機を表すPlayerクラスや敵を表すEnemyクラスといった、独特の機能を持つクラスを作成するには、Characterクラスでは不十分なので、このクラスを拡張したり、書き直すといった作業が必要になります。しかし、一度完成したクラスに「付け足す」「書き直す」ことは、バグを減らすという観点からはなるべく避けるべきです。書き換えると新たなバグ生む危険性があるためです。一度完成したコードを書き換えるのは、細心の注意と多大な労力が必要になります。

C++では、継承(inheritance)という仕組みで、すでに作成したクラスをもとに新しいクラスを効率よく作成できるようになっています。既存のクラスを継承した新しいクラスは、既存のクラスのメンバを「受け継ぐ」ようなかたちになります。既存のクラスには直接さわらずに、新しく必要な属性や動作を付け足すように、新しいクラスのコードを記述することができます。
継承して新しいクラスを作成するには、以下のような書式になります。

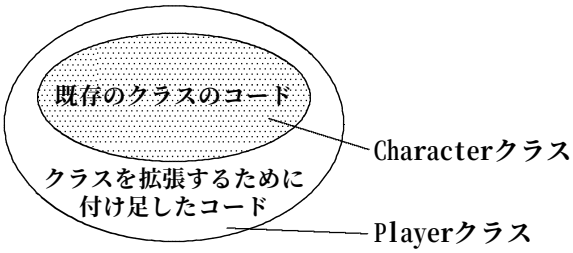
```
class 新しいクラス名 : アクセス指定子 元になるクラス名 {
    追加・変更のメンバを記述
};
```

「元になるクラス」を基底クラス(またはスーパークラス)、新しく拡張されたクラスを派生クラス(またはサブクラス)と呼びます。
派生クラスでは、基底クラスの持つprivate以外のメンバ関数とメンバ変数を、あたかも自分のクラスで記述したかのように扱うことができます。

Characterクラスを継承して「残機」の加わったPlayerクラスを作成するとします。Characterクラスに残機を保持するメンバ変数とそれに付随するメンバ関数をつけ加えればよいでしょう。これは、以下のように定義できます。

```
// キャラクタークラス
class Character {
public:
    // コンストラクタ
    Character() : m_PosX(0.0), m_PosY(0.0), m_AnimeNo(0) {}
    Character(const double pos_x, const double pos_y)
        : m_PosX(pos_x), m_PosY(pos_y), m_AnimeNo(0) {}

    void Move(const double add_x, const double add_y);      // 移動
    void Animetion();                                       // アニメーション

private:
    double   m_PosX;        // ｘ座標
    double   m_PosY;        // ｙ座標
    int      m_AnimeNo;     // アニメーション番号
};

// プレイヤークラス
class Player : public Character {
public:
    Player(const double x, const double y, const int rest);    // コンストラクタ
    int Oneup()   { return ++m_Rest; }                         // １機増やす
    int Onedown() { return --m_Rest; }                         // １機減らす

private:
    int   m_Rest;    // 残機
};
```

こうすると、Characterクラスを内部に含んだPlayerクラスができあがります。この場合、Characterクラスが基底クラス、Playerクラスが派生クラスとなります。



protected

基底クラスCharacterは、すべてのメンバ変数がprivateとして記述されています。privateなメンバは、クラス外からアクセスすることはできません。たとえ派生クラスであっても、直接アクセスすることができないのです。つまり、派生クラスPlayerが移動処理や座標取得を行うためにメンバ変数m_PosXやm_PosYに直接アクセスしようとしても、エラーになってしまいます。

通常、派生クラスと基底クラスは密接な関係があるため、このアクセス制限が不便な場合もあります。このような場合に用いるのがprotectedです。

アクセス指定子には、privateとpublicのほかにprotectedがあり、これを指定すると、そのクラスと派生クラスが直接アクセスできるメンバとなります。派生クラスでも直接アクセスしたいメンバはprotectedにしますが、多用するとカプセル化のメリットを損なうので注意しましょう。

```
// キャラクタークラス
class Character {
public:
    // コンストラクタ
    Character() : m_PosX(0.0), m_PosY(0.0), m_AnimeNo(0) {}
    Character::Character(const double pos_x, const double pos_y)
        : m_PosX(pos_x), m_PosY(pos_y), m_AnimeNo(0) {}

    void Move(const double add_x, const double add_y);      // 移動
    void Animetion();                                       // アニメーション

protected:
    double   m_PosX;        // x座標
    double   m_PosY;        // y座標

private:
    int      m_AnimeNo;     // アニメーション番号
};
```

継承とアクセス指定子

継承では、基底クラスのprivate以外のメンバを、どのようなアクセス制限で継承するのかという指定が必要となります。よく使われるのが、アクセス指定子にpublicを指定し、基底クラスのアクセス制限をそのまま継承するpublic継承です。public継承では、基底クラスのpublicなメンバは派生クラスでもpublicなメンバ、基底クラスのprotectedなメンバは派生クラスでもprotectedなメンバとして継承します。

基底クラスのprivateでないすべてのメンバをprivateなメンバとして継承するprivate継承や、protectedなメンバとして継承するprotected継承といったものもあります。

このように、基底クラスのメンバは、継承の種類によって以下のようにアクセス指定子が変更されて継承されます。なお、アクセス制限の変更は緩やかなものからより厳しいものへの変更のみで、厳しいものからより緩やかなものへ変更すること(protectedをpublicにするなど)はできません。

継承の種類	基底クラスのpublicメンバ	基底クラスのprotectedメンバ	基底クラスのprivateメンバ
public継承	変更なし(public)	変更なし(protected)	アクセスできない
private継承	privateになる	privateになる	アクセスできない
protected継承	protectedになる	変更なし(protected)	アクセスできない

つぎのCharacterクラスを継承してPlayerクラスを作成する場合、継承時に指定するアクセス指定子によって、イメージのように継承されたPlayerクラスが作成されます。

```
// キャラクタークラス
class Character {
public:
    void Move();            // 移動
    void Animetion();       // アニメーション

protected:
    double   m_PosX;        // ｘ座標
    double   m_PosY;        // ｙ座標

private:
    int      m_AnimeNo;     // アニメーション番号
};

// プレイヤークラス
class Player : アクセス指定子 Character {
public:
    int Oneup();            // １機増やす
    int Onedown();          // １機減らす

private:
    int   m_Rest;           // 残機
};
```

```
class Player {
public:
    void Move();        // 継承
    void Animetion();   // 継承

    int Oneup();
    int Onedown();

protected:
    double  m_PosX;     // 継承
    double  m_PosY;     // 継承

private:
    int     m_Rest;

    int     Character::m_AnimeNo;  // アクセス不可
};
```

▲public継承によるPlayerクラスのイメージ

```
class Player {
public:
    int Oneup();
    int Onedown();

private:
    void Move();        // 継承
    void Animetion();   // 継承

    double  m_PosX;     // 継承
    double  m_PosY;     // 継承

    int     m_Rest;

    int     Character::m_AnimeNo;  // アクセス不可
};
```

▲private継承によるPlayerクラスのイメージ

```
class Player {
public:
    int Oneup();
    int Onedown();

protected:
    void Move();        // 継承
    void Animetion();   // 継承

    double  m_PosX;     // 継承
    double  m_PosY;     // 継承

private:
    int     m_Rest;

    int     Character::m_AnimeNo;  // アクセス不可
};
```

▲protected継承によるPlayerクラスのイメージ

継承とコンストラクタ

コンストラクタは、インスタンスが生成されるときに呼び出される特別なメンバ関数であり、メンバ変数などの初期化を行います。派生クラスのインスタンスを作成する場合には、何らかの方法で基底クラスのコンストラクタを呼び出し、基底クラスが持つメンバ変数を初期化しなければなりません。

基底クラスのコンストラクタは、派生クラスから自動的に呼び出されるようになっています。派生クラスのコンストラクタが呼び出されると、何も記述されていない場合でも、最初に基底クラスのコンストラクタが呼び出され、実行されます。基底クラスのコンストラクタの処理がすべて終了したあと、派生クラスのコンストラクタが実行されます。

派生クラスのコンストラクタで基底クラスのコンストラクタが明示されていない場合、基底クラスのデフォルトコンストラクタが暗黙のうちに呼び出されます。基底クラスのデフォルトでないコンストラクタを呼び出すには、派生クラスのコンストラクタ初期化子を用います。
たとえば、派生クラスPlayerのコンストラクタから基底クラスCharacterの引数付きコンストラクタを呼び出すには、以下のように記述します。

```
Player(const double x, const double y, const int rest) : Character(x, y), m_Rest(rest);
```

継承とデストラクタ

デストラクタは、インスタンスが破棄されるときに呼び出される特別なメンバ関数であり、インスタンスを正しく解放するための処理を記述します。派生クラスのインスタンスが破棄される場合には、何らかの方法で基底クラスのデストラクタを呼び出し、基底クラスを正しく解放しなければなりません。
基底クラスのデストラクタは、派生クラスから自動的に呼び出されるようになっています。派生クラスのデストラクタが実行されたあと、基底クラスのデストラクタが呼び出され、実行されるようになっています。

is-a関係とhas-a関係

継承された２つのクラス(基底クラスと派生クラス)の関係のことをis-a関係またはa-kind-of関係と呼びます。これは、「派生クラス is a(kind of) 基底クラス」すなわち「派生クラスは基底クラス(の一種)である」という関係です。オブジェクト指向設計において、２つのクラスの関係が「○○は△△である」と考えられる場合には、継承を使ってクラスを関連付けます。

継承に対し、あるオブジェクトをいくつかのオブジェクトの組み合わせとして実現したり、あるオブジェクトの一部を別のオブジェクトで構成したりする方法を集約(aggregation)と呼び、この関係が特に強いものを合成(composition)と呼びます。
また、構成要素のオブジェクトとそれらを組み合わせてできたオブジェクトとの関係を、has-a関係またはpart-of関係と呼びます。これは、「クラスA has a(part of) クラスB」すなわち「クラスAはクラスBを持っている(またはその一部である)」という関係です。オブジェクト指向設計において、２つのクラスの関係が「他方がもう一方を持っている」と考えられる場合には、集約を使ってクラスを関連付けます。

プログラムを構成するすべてのクラスが、継承または集約のいずれかで必ず関連付けられるわけではありません。継承も集約も使わず、単独のクラスとなる場合もあります。継承や集約は、プログラムの構造を整理し、効率的に作成するための手段にすぎません。継承や集約を使うかどうかは、プログラム設計者の判断によります。

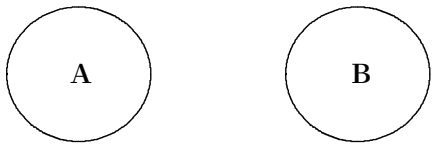
練習問題

1　以下の文章(１)～(６)に示したClassAクラスとClassBクラスの関係が「is-a関係」の場合はA,「has-a関係」の場合はB, どちらでもない場合はCを付けましょう。

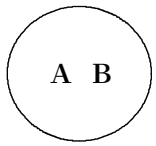
(１)ClassAクラスは、ClassBクラスを継承している。
(２)ClassAクラスのメンバ関数内で、ClassBクラスのオブジェクトを作成した。
(３)ClassAクラスのメンバ変数に、ClassBクラスをデータ型とした変数がある。
(４)ClassAクラスとClassBクラスのメンバ変数に、別の同じクラスをデータ型とした変数がある。
(５)ClassAクラスとClassBクラスは、同じ基底クラスから継承されている。
(６)ClassAクラスは、ClassBクラスの一種だと考えられる。

2　プログラムで使用されるオブジェクトが以下の関係のとき、どのようなクラスを作成するのが望ましいか検討せよ。

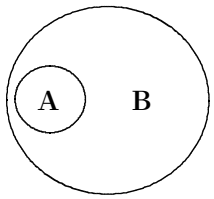
(1)オブジェクトAとBは分離



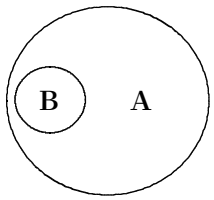
(2)オブジェクトAとBは一致



(3)オブジェクトAはBに含まれる



(4)オブジェクトBはAに含まれる



(5)オブジェクトAとBは一部重なる