Chapter-2 継承

C++の強力な機能の一つに継承というものがあります。この継承という考え方はオブジェクト指向を用いた設計、デザインパターンを学ぶ上で非常に重要な概念になります。このチャプターでは継承について見ていきましょう。

2.1 メンバ変数の継承

まずは、簡単な例で継承を見ていきましょう。例えばレースゲームを作成していることを考えてみて下さい。車には色々な車種があります。ワゴンR、フィット、ヴィッツ、フェラーリ、ポルシェなどなど。これらは当然車種ごとに、ステアリング性能、加速性能、燃費、車体フレームなど異なる点が多数存在します。しかし、どの車種も車であることに違いはありません。そのため共通点がいくつか存在します。タイヤは４つ付いていますし、ハンドルも付いています。アクセル、ブレーキなども付いているはずです。これらの共通部分を抽出して下記のようなクラスを作成します。

```
//車の基底クラス。
class CarBase{
private:
   Tire      tire[4];         //タイヤ
   Handle    handle;      //ハンドル
   BrakePedal  brakePedal;       //ブレーキペダル
   AxelPedal     axelPedal;    //アクセルペダル。
};
```

そして、各車種はCarBaseクラスを継承して実装します。

```
//フィット
class Fit : public CarBase{
Model model;  //フィットの車体モデル。
};
//ワゴンR
class WagonR : public CarBase{
  Model model;  //ワゴンRの車体モデル。
};
//フェラーリ
class Ferrari : public CarBase{
  Model model;  //フェラーリの車体モデル。
};
```

このように記述を行うことで、各車種はCarBaseクラスを継承することができます。そして、各クラスはメンバ変数として、tire[4]、handle、brakePedal、axelPedalを保持することになります。

2.2 メンバ関数の継承

継承はメンバ変数のみではなく、メンバ関数も継承することができます。先ほどの車を例にして見ていきましょう。車には走る処理のRun関数、窓を開けるOpenWindow関数、ドアを開けるOpenDoor関数などなど、いくつも共通の処理が存在するはずです。C++では、このような処理を基底クラスのメンバ関数として記述することで、どの車でも共通の処理として定義することができます。車の基底クラスは次のようになるでしょう。

```
class CarBase{
// 派生クラスでアクセスしたい場合はアクセス指定子をprotectedにする。
protected:
   Tire      tire[4];         //タイヤ
   Handle    handle;      //ハンドル
   BrakePedal  brakePedal;        //ブレーキペダル
   AxellPedal    axellPedal;      //アクセルペダル。
public:
    //走る処理
    void Run();
    //窓開ける。
    void OpenWIndow();
    //ドア開ける
    void OpenDoor();
};
```

この基底クラスを継承した派生クラスはメンバ関数として、Run、OpenWIndow、OpenDoorを保持するようになります。

3.3 継承すべきか委譲すべきか

オブジェクト指向のクラス設計において、車の例のような共通処理のクラス化は継承の他に委譲というテクニックが存在します。では継承と委譲の使い分けはどのようにすればいいのか？ この指針としてよく言われるものに下記のようなものがある。

クラス間の関係がis-aの場合は継承、has-aの場合は委譲。

is-aとは、「フェラーリは車である」のように、派生クラス=基底クラスが成り立つ場合のことを言います。has-aの場合は「フェラーリはブレーキペダルを持っている」という場合になります。 is-aの場合は継承を行うことを検討する、has-aの場合は委譲を行うことを検討してみることが設計の指針になります。

3.4 仮想関数

さて、先ほどのドアを開ける処理ですが、車種によってはスライドドアのものがあれば、従来の引手のドアもあるでしょう。そのため基底クラスのOpenDoor関数に引手のドアの処理を記述している場合、問題が出てきます。もちろんif文などで処理を分けてもいいでしょう。しかしC++であればこれを仮想関数というものを使用することによって、スマートに解決することができます。仮想関数とは派生クラスで実装を変更できる関数になります。仮想関数の実装は下記のようになります。

```
class CarBase{
private:
 Tire        tire[4];         //タイヤ
 Handle      handle;      //ハンドル
```

```
 BrakePedal     brakePedal;        //ブレーキペダル
AxellPedal   axellPedal;     //アクセルペダル。
public:
 //走る処理
 void Run();
//窓開ける。
 void OpenWIndow();
//ドア開ける
 virtual void OpenDoor();
};
//デリカ
class Delica : public CarBase{
Model model;      //デリカの車体モデル。
void OpenDoor();  //デリカはスライドドアなので、オーバーライドする！
};
//ワゴンR
class WagonR : public CarBase{
  Model model;  //ワゴンRの車体モデル。
};
//フィット
class Fit : public CarBase{
  Model model;  //フェラーリの車体モデル。
};
```

ワゴンRとFITは引きドアなので、CarBaseに実装されているデフォルトのOpenDoor関数を使用しています。しかしデリカはスライドドアのため、OpenDoorをオーバーライドしています。基底クラスの仮想関数の実装を派生クラスで再定義することをオーバーライドといいます。

さて、実はここまでの話には少し嘘があります。実は仮想関数なんて使わなくても、実はオーバーライドは可能です。仮想関数が真価を発揮するのはポリモーフィズム(多態性)と言われる機能を使うときになります。では次の節では多態性について見ていきましょう。

3.4.1 仮想関数を使わなくてもオーバーライドできるって本当？

Sample_03_01を使って確認してみましょう。

3.5 ポリモーフィズム(多態性)

「基底クラスのポインタ型の変数に、派生クラスのインスタンスのアドレスを代入すると、あたかも派生クラスのインスタンスであるかのように振舞う」ことをいいます。ではサンプルコードを見てみましょう。

```
class HogeBase{
 public:
    //仮想関数版のPrint関数
    virtual void Print()
    {
       std::cout << “HogeBase\n”;
    }
 };
 class Hoge : public HogeBase{
 public:
```

```
    void Print()
    {
      std::cout << “Hoge\n”;
    }
 };
int main()
{
 Hoge hoge;
 HogeBase* hogeBase = &hoge;  //HogeBase型のポインタ変数にhogeのアドレスを代入。
 hogeBase->Print();           //Hogeと表示される。これがポリモーフィズム。
 return 0;
}
```

このように、HogeBaseを継承しているHogeクラスのインスタンスは、基底クラスのHogeBaseのポインタ型の変数にアドレスを代入できます。そして、HogeBase型のポインタはPrint関数を呼び出すと、あたかもHogeであるかのように振る舞います。Print関数が仮想関数でない場合は、HogeBaseと表示されます。

3.5.1 ポリモーフィズムを使う理由。

では、前節のレースゲームを例にして考えてみましょう。レースゲームではユーザーがレースを始める前に自分が操作する車を選択します。そして、ゲーム中のアップデート関数では選択した車に対する操作(ブレーキやアクセルやドアを開くなど)が実行されるはずです。では、ポリモーフィズムを知らない不幸なコードを見てみましょう。

```
//car.h
//車の基底クラス
class CarBase{
private:
 Tire        tire[4];         //タイヤ
 Handle      handle;      //ハンドル
 BrakePedal     brakePedal;        //ブレーキペダル
AxellPedal   axellPedal;      //アクセルペダル。
public:
 //ブレーキをかける処理
 virtual void Brake();
//アクセル
 virtual void Accell();
//走る処理
 virtual void Run();
};
//デリカ
class Delica : public CarBase{
//ブレーキをかける処理
  void Brake();
//アクセル
  void Accell();
};
//ワゴンR
class WagonR : public CarBase{
   //ブレーキをかける処理
   void Brake();
```

```
};
//フィット
class Fit : public CarBase{
//アクセル
   void Access();
};

  .
  .
  .
//car.cpp
#include “car.h”
Int selectCarType;  //0だとデリカ、1だとワゴンR、２だとFIT
Delica delica;
WagonR wagonR
Fit fit;

//ブレーキの処理。
void Brake()
{
  If(selectCarType == 0){
    //デリカ
    delica.Brake();
  }else if(selectCarType == 1){
    //ワゴンR
    wagonR.Brake();
  }else if(selectCarType == 2){
    //フィット
    fit.Brake();
  }
}
//アクセルの処理。
void Accell()
{
  If(selectCarType == 0){
    //デリカ
    delica.Accell();
  }else if(selectCarType == 1){
    //ワゴンR
    wagonR.Accell();
  }else if(selectCarType == 2){
    //フィット
    fit.Accell();
  }
}
//走る処理。
void Run()
{
  If(selectCarType == 0){
   delica.Run();
  }else if(selectCarType == 1){
    wagonR.Run();
  }else if(selectCarType == 2){
    fit.Run();
```

```
  }
}
//メイン関数
int main()
{
  std::cin >> selectCarType;
  while(true){  //ゲームループ。
    //ブレーキの処理
    Brake();
  //アクセルの処理。
    Accell();
    //走る処理。
    Run();
  }
}
```

ポリモーフィズムを知らないプログラマはこのようなコードを書くと思います。実際のレースゲームであれば、車種はもっと多いはずなのでselectCarTypeを使用したif文の数は100を軽く超えることになるでしょう。そして。この不幸なプログラマは下記のような仕様変更が発生した時に定時で帰ることはできなくなるでしょう。

「クライアントから車のドアを開けられるようにして欲しいという要望が来たので対応してください。」

このコードを書いたプログラマはOpenDoorという処理を記述して、また新しくselectCarTypeの条件文を追加して、OpenDoorという関数の呼び出しを100箇所以上記述することになります。

あなたはAccell関数、Brake関数、Run関数に新しいselectCarTypeを使用する条件文を記述して、各種メンバ関数の呼び出しコードを記述することになります。そして、ある日このプログラマは、また新しい車種が追加されたときにRun関数だけコードを追加することを忘れてしまって、不具合に頭を悩ませることになるでしょう。

では、このプログラムをポリモーフィズムを使用するプログラムで書き換えてみましょう。ヘッダーファイルに変更点はありません。

```
//car.cpp
#include “car.h”
Int selectCarType;  //0だとデリカ、1だとワゴンR、2だとFIT
Delica delica;
WagonR wagonR
Fit fit;

CarBase* carBaseArray[3];  //CarBaseのポインタ型の配列
//ブレーキの処理。
void Brake()
{
  carBaseArray[selectCarType]->Brake();  //これがポリモーフィズム！！！
}
//アクセルの処理。
void Accell()
{
  carBaseArray[selectCarType]->Accell();  //これがポリモーフィズム！！！
}
//走る処理。
void Run()
```

```
{
  carBaseArray[selectCarType]->Run();  //これがポリモーフィズム！！！
}
//メイン関数
int main()
{
  carBaseArray[0] = & delica;      //派生クラスのインスタンスアドレスを基底クラスのポインタ型の変数に代入。
  carBaseArray[1] = & WagonR;  //派生クラスのインスタンスアドレスを基底クラスのポインタ型の変数に代入。
  carBaseArray[2] = &Fit;         //派生クラスのインスタンスアドレスを基底クラスのポインタ型の変数に代入。
  std::cin >> selectCarType;
  while(true){  //ゲームループ。
    //ブレーキの処理
    Brake();
  //アクセルの処理。
    Accell();
    //走る処理。
    Run();
  }
}
```

非常にシンプルな短いコードになりました。これがポリモーフズムを活用したプログラムになります。ポリモーフズムとは同じ操作で、異なる動作をするものとなります。このようなコードを記述したプログラマであれば、新しい車種が追加された場合、追加で記入するプログラムは下記のたった一行になります(もちろん新しい車種のクラスは作成しますが)。

```
int main()
{
  carBaseArray[0] = & delica;      //派生クラスのインスタンスアドレスを基底クラスのポインタ型の変数に代入。
  carBaseArray[1] = & WagonR;      //派生クラスのインスタンスアドレスを基底クラスのポインタ型の変数に代入。
  carBaseArray[2] = &Fit;          //派生クラスのインスタンスアドレスを基底クラスのポインタ型の変数に代入。
  carBaseArray[3] = &vitz          //VITZを追加！
  std::cin >> selectCarType;
  while(true){  //ゲームループ。
    //ブレーキの処理
    Brake();
  //アクセルの処理。
    Accell();
    //走る処理。
    Run();
  }
}
```

Brake関数、Accell関数、Run関数に新しい条件文を記述する必要は全くありません。なぜならば、vitzのインスタンスのアドレスを代入されたcarBaseArrayはあたかもでvitzであるかのように振舞うからです。

新しいOpenDoorという関数が追加されても下記の処理の追加だけで完了します。

```
carBaseArray[selectCarType]->OpenDoor();
```

　ポリモーフィズムを上手に活用したプログラムは関数呼び出しの追加忘れや仕様変更に非常に強いプログラムになり、ヒューマンエラーの発生を大きく下げてくれます。