```
    Model model;                    //デリカの車体モデル。
    void OpenDoorSlideDoor();   //スライドドア用のドアを開く関数を別名定義。
};
```

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。

3.5 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。

1 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。

2

```
//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();
  };
```

3

4

```
//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。
また、次のような要望が来た時も悲しいことになります。

「クライアントから新しい車種を追加して欲しいという要望が来たので対応してください。」

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。

Chapter 4 std::vector