C++勉強会

第2回 クラスの継承と多態性

Created by T.Miyaji and F.Hirakoba

⚠️ このスライドは、未完成です

完成版は、後日うpします

目次

- クラスの作り方（復習）
- クラス図
- クラスの責務
- 合成
- 継承
- 仮想関数と純粋仮想関数

クラスの作り方（復習）

1. 作りたいクラス名に沿ってヘッダファイルを作成する。
2. ヘッダファイルにインクルードガードを作る。
3. クラスの宣言とコンストラクタを実装する。
4. ヘッダファイル名と同名のソースファイルを作成する。
5. クラスを実装する。

以降は、手順1つひとつの詳細について述べます。

クラスの作り方（復習）

1. 作りたいクラス名に沿ってヘッダファイルを作成する。

- 宣言するクラス名が`ClassName`のとき、ヘッダファイル名は`class_name.h`とします。
- `string`や`vector`といったライブラリは、ヘッダファイルに記述します。

クラスの作り方（復習）

2. ヘッダファイルにインクルードガードを作る。

インクルードガードとは、ヘッダファイルを1回だけ呼ばれるようにするテクニックのことです。

```
#ifndef CLASS_NAME_H
#define CLASS_NAME_H

// ここにクラスの宣言を書く

#endif
```

- `#ifndef CLASS_NAME_H`とは、`CLASS_NAME_H`が定義されていなければ真となる条件文です。クラス名が`ClassName`のときは、`CLASS_NAME_H`のようにすべて大文字で指定します。
- `#define CLASS_NAME_H`とは、`CLASS_NAME_H`を定義するという文です。これによって2回目以降は、`#ifndef`の条件式が偽となります。
- `#endif`は、`#ifndef`のブロックの終了を表します。

クラスの作り方（復習）

3. クラスの宣言とコンストラクタを実装する。

```
#ifndef CLASS_NAME_H
#define CLASS_NAME_H

class ClassName {
  private:
    int x;
  public:
    ClassName() { }
    explicit ClassName(int x_) : x(x_) { }
    void func();
};

#endif
```

- 引数を取るコンストラクタを作る場合は、**引数を取らないコンストラクタを必ず作成**してください。これはクラスから作ったオブジェクトをデータ構造にもつときに不都合があるからです。
- 引数を取るコンストラクタを作る場合は、`explicit`指定子をつけてください。これは、暗黙的なキャストが行なわれないようにするためです。
- メンバ関数は、ヘッダファイルでは実装しません。

クラスの作り方（復習）

4. ヘッダファイル名と同名のソースファイルを作成する。

- 宣言するクラス名が`ClassName`のとき、ヘッダファイル名は`class_name.cpp`とします。

クラスの作り方（復習）

5. クラスを実装する。

```
#include "class_name.h"

void ClassName::func()
{
  // 関数の中身
}
```

- ソースファイルでは、実装するクラスの宣言が書かれたヘッダファイルのみ読み込みます。
- メンバ関数は、`クラス名::関数名`で指定します。これは、クラスというスコープ内の関数を実装するためです。

クラス図

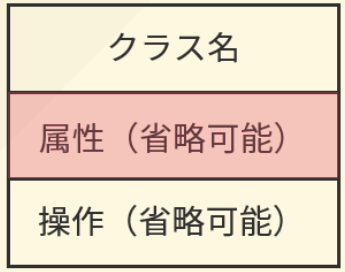
クラス図とは、クラスとクラス間の関係を表現する静的な構造図。

クラス図

属性の記述例と記法

- 属性の記述例

```
- name : string
- num : int = 0
```



- 属性の記法（一部）

可視性　名前：型 ＝ 初期値

※ 可視性「-」は、C++では private に対応します。

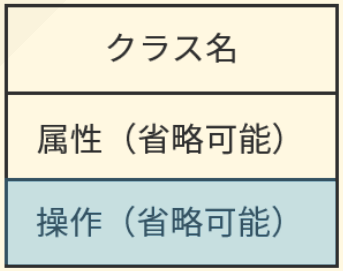
※ クラス図では、属性または属性の一部を省略することがあります。

クラス図

操作の記述例と記法

- 操作の記述例

```
+ func(value : int) : void
```



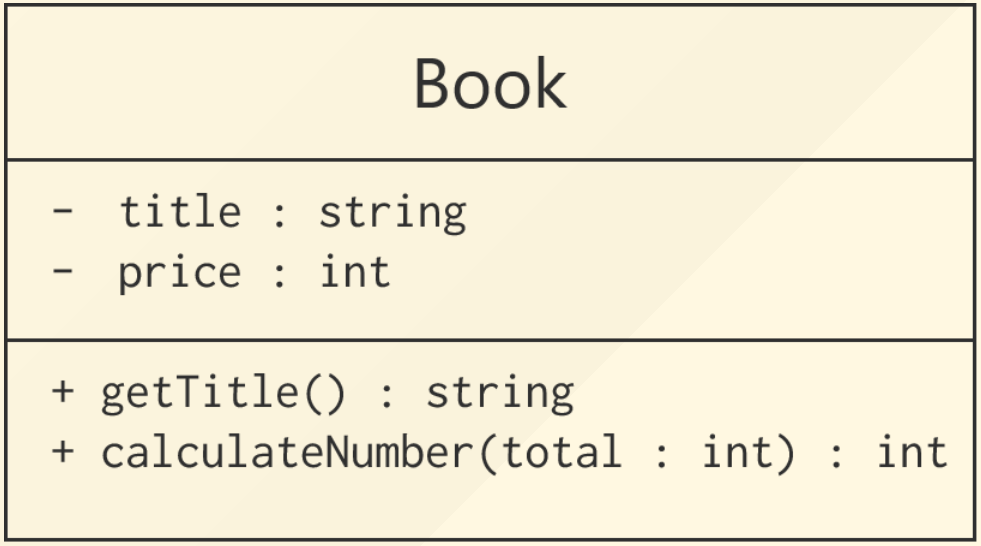
- 操作の記法（一部）

可視性　名前（引数名 : 引数の型）: 戻り値の型

※ 可視性「+」は、C++では public に対応します。

※ クラス図では、属性または属性の一部を省略することがあります。

Q2-1 次のクラス図からクラスを作ってみよう



- ソースコードは、こちらにあります。ここで、Makefileという名前のファイルを作業ディレクトリにダウンロードしてください。
- `calculateNumber()`は、価格の合計から本の冊数を求めるメンバ関数。
- コンパイルは、作業ディレクトリで`make`と入力するとできます。
- 出力結果が以下のようになればOK
  (出力) 吾輩は猫である:2040円の冊数は3冊です。

クラスの責務

1つのクラスは、必ず1つの 責務 をもつ。

- 責務 = 変更理由
- 変更する理由が複数あるクラスは、分割すべき
- 特に変更する理由がなければ、クラスの分割はしない

クラスの責務

複数の責務をもつクラスの例

```
class BookList {
  private:
    std::vector<Book> Books;
  public:
    BookList() { }
    int calculate(); // 本の価格の合計を計算する
    void output();   // 本の題名と価格を出力する
```

クラスの責務

複数の責務をもつクラスの例

```
class BookList {
  private:
    std::vector<Book> Books;
  public:
    BookList() { }
    int calculate(); // 本の価格の合計を計算する
    void output();   // 本の題名と価格を出力する
};
```

もし、`output()`の出力をHTMLやCSVにしたいときは?

➡️ BookListがデータの出力に対して責務をもつことは不適切!

クラスの責務

複数の責務をもつクラスの分割例

```
class BookList {
  private:
    std::vector<Book> Books;
  public:
    BookList() { }
    int calculate(); // 本の価格の合計を計算する
};

class Outputter {
  private:
    BookList list;
  public:
    Outputter() { }
    explicit Outputter(BookList& list_) : list(list_) { }
    void HTML();
    void CSV();
};
```

合成

- 合成とは、「あるクラスが他のクラスのオブジェクトを保持していること」を指す
- 合成には次の2種類がある
  - has-a関係（AがBをもっている）
  - is-implemented-in-terms-of関係（AはBを用いて実装している）

合成

has-a関係（AがBをもっている）

```
class A {
  private:
    B b;
  public:
    A() { }
    explicit A(B& b_) : b(b_) { }
    void funcA();
};
```

Aのメンバ関数内でBのメンバ関数を呼び出すことができる。
クラスの責務分割で **合成** は、非常に役に立つ考え方！

⚠️ メンバ関数やコンストラクタの引数に、クラスのオブジェクトをとる場合は、**参照渡しにしよう!** (コピーコストを抑えるため)

(実は、`std::string`もクラスなので参照渡しのほうがよい)

合成

is-implemented-in-terms-of関係（AはBを用いて実装している）

```
class A { ... };

void A::funcA()
{
  B b;  // Aのメンバ関数内でBを宣言する
  b.funcB();
}
```

上記のようなis-implemented-in-terms-of関係は、`A::funcA()`がBに依存している。

合成

is-implemented-in-terms-of関係（AはBを用いて実装している）

```
class A { ... };

void A::funcA()
{
  B b;  // Aのメンバ関数内でBを宣言する
  b.funcB();
}
```

上記のようなis-implemented-in-terms-of関係は、`A::funcA()`がBに依存している。

➡️ `A::funcA()`の単体テストもBに依存してしまう。

- Bを作るまで、Aはテストできない。
- Bの実装を変えれば、Aのテストの結果は変わる。

よって、このような実装はできるだけ避けること！

合成

is-implemented-in-terms-of関係（AはBを用いて実装している）

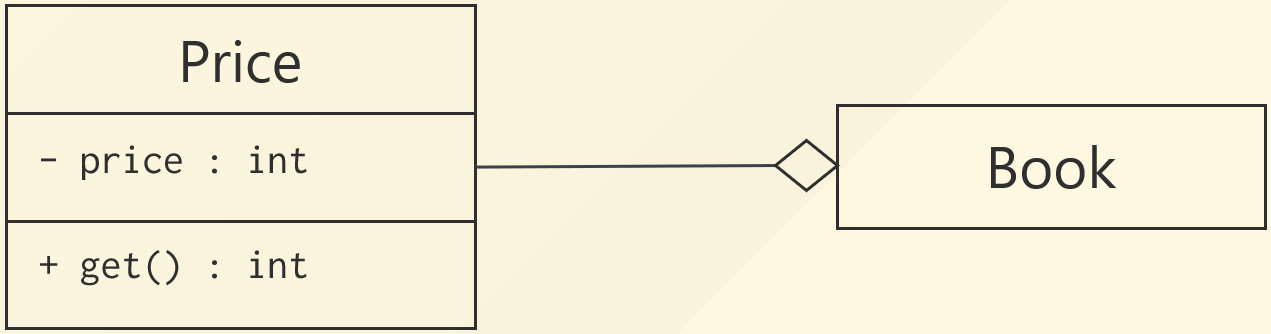
```
class A { ... };

void A::funcA()
{
  B b;  // Aのメンバ関数内でBを宣言する
  b.funcB();
}
```

上記のようなis-implemented-in-terms-of関係は、`A::funcA()`がBに依存している。

➡️ Aのコンストラクタまたは`A::funcA()`の引数にBクラスのオブジェクトを取るように改良しよう（依存性注入）。

Q2-2 Priceクラスを作って、Bookクラスと合成してみよう



- ダイヤ型の矢印は、合成を意味する。(BookがPriceを保持している)
- ソースコードは、こちらにあります。Makefileもダウンロードしてください。
- `get()`は、メンバ変数`price`を返すメンバ関数。
- Priceクラスを作成し、BookクラスにPriceクラスのオブジェクトを持たせてみてください。
- コンパイルは、作業ディレクトリで`make`と入力するとできます。
- 出力結果が以下のようになればOK
  (出力) 吾輩は猫である:2040円の冊数は3冊です。

継承

- 継承とは、「あるクラスの特性を受け継いで新たなクラスを作ること」を指す。
- 継承元のクラスを 基本クラス とよび、基本クラスを受け継ぐクラスを 派生クラス とよぶ。
- 継承関係にあるクラス同士を**is-a関係がある**という。

継承

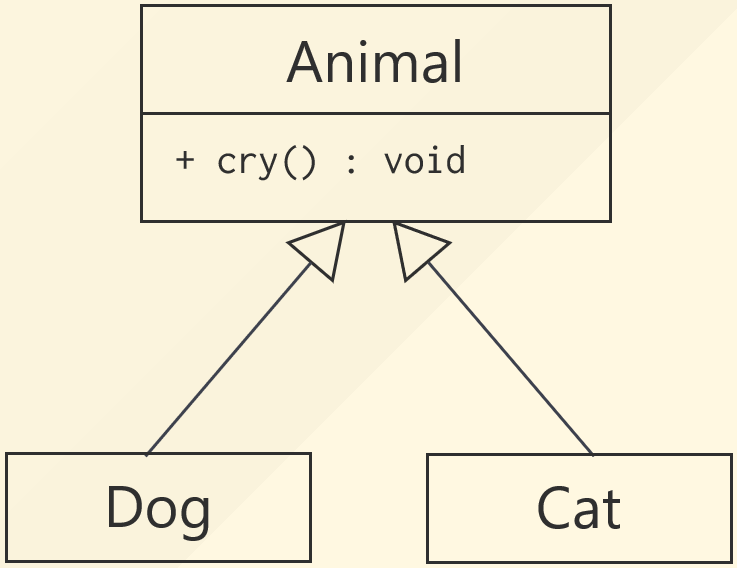
C++による継承の書き方

```
class A {
  private:
    int x;
  public:
    ...
    void funcA();
};

class B : public A {
  private:
    int y;
  public:
    B() : A() { }
    explicit B(int y_) : A(), y(y_) { }
    ...
    void funcB();
};
```

クラスBは、クラスAを受け継ぐため、公開しているAのメンバ関数を扱える。
ただし、非公開`private`のメンバ変数は受け継げない。

Q2-3 Animalクラスを継承してDogクラスとCatクラスを作ってみよう



- Dog/Catクラスは、Animalクラスを継承する(三角矢印は、継承を表す)
- Animalクラスのソースコードはこちらにあります。
- Dog/Catクラスで受け継いだ`cry()`関数を再定義できる(オーバーライドという)
- Dog/Catクラスで`cry()`を実装する(例 Dogなら「わんわん」と鳴く)。

継承

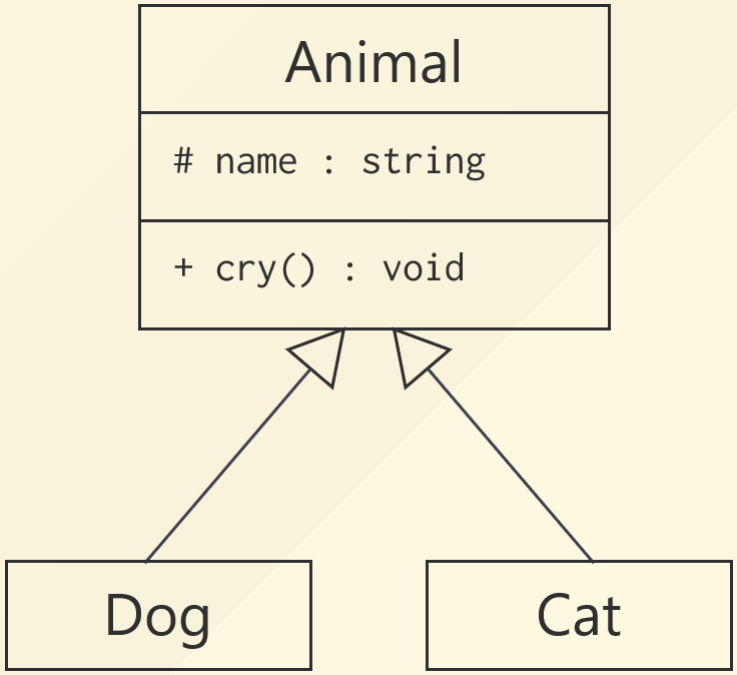
変数を受け継がせたいとき ➡️ `protected`指定子を使う

```
class A {
  protected:
    int x;
  public:
    A() { }
    explicit A(int x_) : x(x_) { }
    ...
};

class B : public A {
  public:
    B() : A() { }
    explicit B(int x) : A(x) { }
    ...
};
```

- `protected`指定子をつけたメンバ変数は、基本クラスと派生クラスでのみ使用できる。
- `protected`指定子は、メンバ関数につけることもできる。

Q2-4 Animalクラスに名前を追加してみよう



- #は、`protected`指定子を表す。
- Dog/Catクラスの`cry()`で`name`を呼び出してみよう。

継承

継承の利点

- 基本クラスのメンバ変数やメンバ関数を受け継ぐことができる。
  - 派生クラスは、基本クラスで定義していない残りを実装すればよい(差分プログラミング)。
  - 基本クラスのコードを再利用できる。
- 多態性を実現できる。
  - `cry()`関数は、DogオブジェクトもCatオブジェクトも持っているが、オブジェクトによって振る舞いが違う。

継承には上記のような利点があるが、正しく使わないと**思わぬバグ**を生むことがある。

継承

つづく。