継承とポリモーフィズム

- 親クラスと子クラス

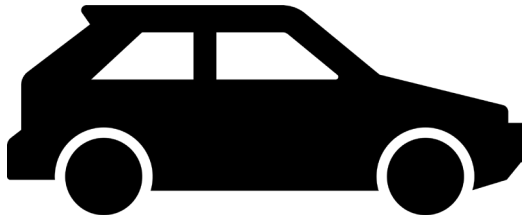
元となるクラスの性質を引き継ぎつつ、新しい機能を付け足して、クラスを拡張していくことをクラスの**継承**という

元となるクラスは、**親クラス**（基底クラス、スーパークラス）と呼び、継承先のクラスを**子クラス**（派生クラス、サブクラス）と呼ぶ

継承

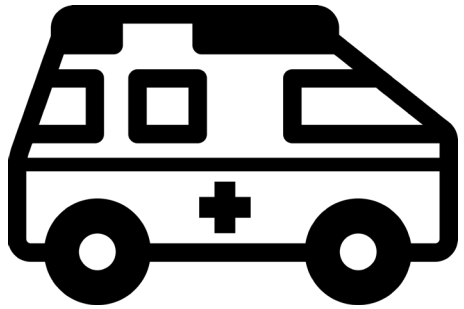
- 教科書P176~179 **Sample501**

親クラス:Car



継承

子クラス:Ambulance



- C++作業フォルダ内に**Sample501**フォルダを作成

```
robocopy Sample401 Sample501
cd Sample501
copy car.h ambulance.h
copy car.cpp ambulance.cpp
```

継承

- car.h (Sample501)

```
#pragma once
class Car {
public:
    Car();
    virtual ~Car();
    void setSpeed(double speed);
    double getSpeed();
    double getMigration();
    void drive(double hour);
private:
    double m_speed;
    double m_migration;
};
```

継承

- car.cpp (Sample501)

変更なし

継承

- ambulance.h (Sample501)

```
#pragma once
#include “car.h”
class Ambulance: public Car {
public:
    Ambulance();
    virtual ~Ambulance();
    void sevePeople();
private:
    int m_number;
};
```

継承

- ambulance.cpp (Sample501)

```
#include “ambulance.h”
#include <iostream>
using namespace std;
Ambulance::Ambulance() : m_number(119) {
    cout << “Ambulanceクラスのインスタンス生成” << endl;
}
Ambulance::~Ambulance() {
    cout << “Ambulanceクラスのインスタンス消去” << endl;
}
void Ambulance::sevePeople(){
    cout << “救急救命活動” << endl
      << “呼び出しは” << m_number << “番” << endl;
}
```

継承

- main.cpp (Sample501)

```
int main() {
    cout << “Carクラスの処理” << endl;
    Car* pkuruma = new Car();
    pkuruma->setSpeed(40);
    pkuruma->drive(1.5);
    cout << “総移動距離:” << pkuruma->getMigration()
         << “km” << endl;
    delete pkuruma;
```

継承

- main.cpp (Sample501)

```
    cout << “Ambulanceクラスの処理” << endl;
    Ambulance* pAmb = new Ambulance();
    pAmb->setSpeed(60);
    pAmb->drive(2);
    cout << “総移動距離:” << pAmb->getMigration()
         << “km” << endl;
    pAmb->sevePeople();
    delete pAmb;
    return 0;
}
```

継承

- コンパイルの仕方

  コマンドプロンプトで次のコマンドを入力する

```
cl /EHsc main.cpp car.cpp ambulance.cpp
```

※すべてのcppファイルを列挙してください

  成功したら、main.exeを実行して結果を確認

継承

- car.h (Sample501)

```
#pragma once
class Car {
public:
    Car();
    virtual ~Car();
    void setSpeed(double speed);
    double getSpeed();
    double getMigration();
    void drive(double hour);
private:
    double m_speed;
    double m_migration;
};
```

デストラクタの前にvirtual修飾子を付ける(テキストP.200)

継承

- ambulance.h (Sample501)

```
#pragma once
class Ambulance: public Car {
public:
     Ambular
     virtual
     void se
private:
     int m_n
};
```

クラスの継承

class 子クラス名 : public 親クラス名

クラスを継承することで、親クラスの持つpublicなメンバを子クラスで使用することができる

また、子クラス独自のメンバを追加して、クラスの機能を拡張することも可能

継承

- ambulance.h (Sample501)

```
#pragma once
class Ambulance: p
public:
     Ambulance();
     virtual ~Ambulance();
     void sevePeople();
private:
     int m_number;
};
```

デストラクタの前にvirtual修飾子を付ける(テキストP.200)

継承

- ambulance.h (Sample501)

```
#pragma once
class Ambulance: public Car {
public:
     Ambulance();
     virtual ~Ambulance();
     void sevePeople();
private:
     int m_number;
};
```

子クラスで追加した
メンバ関数 sevePeople()
メンバ変数 m_number

注意！！
子クラスで定義したものは親クラスからはアクセス不可

継承

- ambulance.cpp (Sample501)

コンストラクタで
メンバ変数 m_number を 119 で初期化

```
#include “ambulanc
#include <iostream
using namespace std;
Ambulance::Ambulance() : m_number(119) {
    cout << “Ambulanceクラスのインスタンス生成” << endl;
}
Ambulance::~Ambulance() {
    cout << “Ambulanceクラスのインスタンス消去” << endl;
}
void Ambulance::sevePeople(){
    cout << “救急救命活動” << endl
      << “呼び出しは” << m_number << “番” << endl;
}
```

継承

- ambulance.cpp (Sample501)

```
#include “ambulance.h”
#include <iostream>
using namespace std;
Ambulance::Ambulance() : m_number(119) {
    cout << “Ambulanceクラスのインスタンス生成” << endl;
}
Ambulance::~Ambulance() {
    cout << “Ambulanceクラスのインスタンス消去” << endl;
}
void Ambulance::sevePeople(){
    cout << “救急救命活動” << endl
      << “呼び出しは” << m_number << “番” << endl;
}
```

sevePeople関数の実装

継承

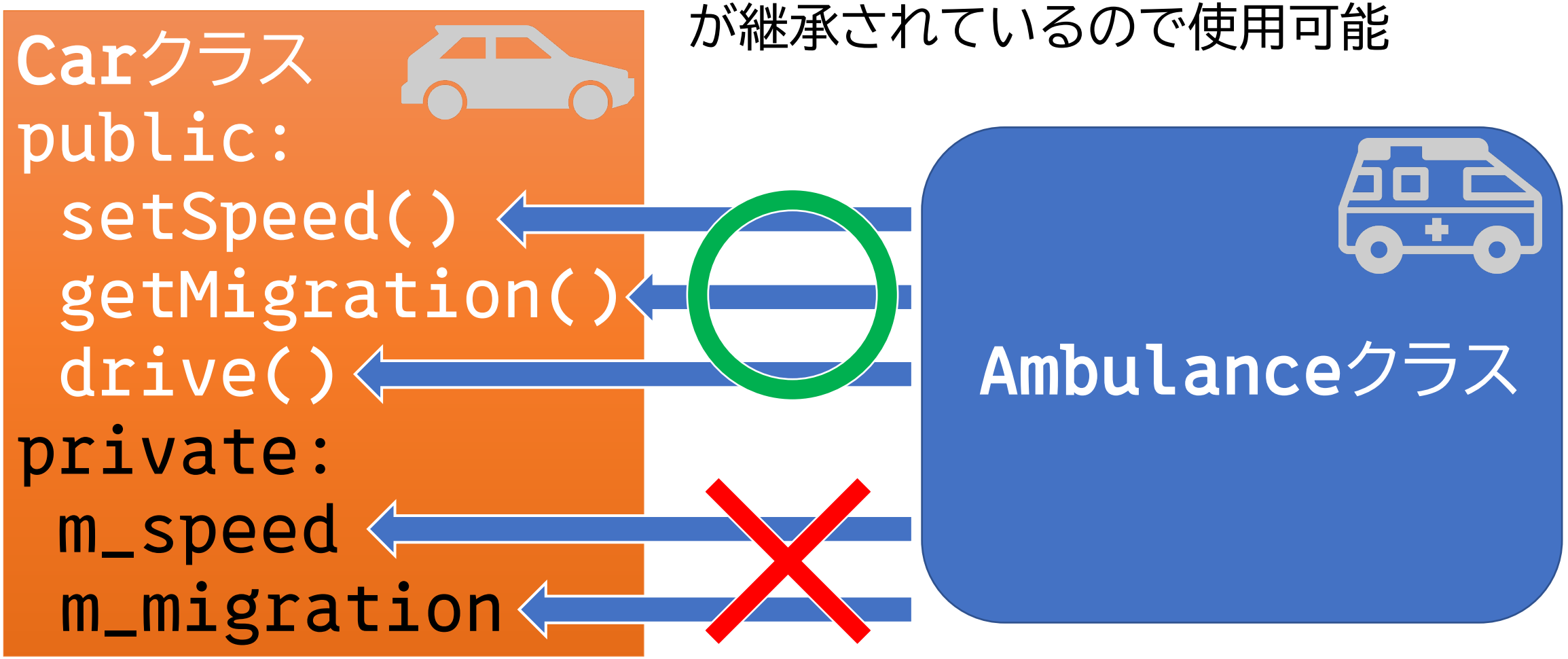
- main.cpp (Sample501)

```
    cout << “Ambulanceクラスの処理” << endl;
    Ambulance* pAmb = new Ambulance();
    pAmb->setSpeed(60);
    pAmb->drive(2);
    cout << “総移動距離:” << pAmb->getMigration()
         << “km” << endl;
    pAmb->sevePeople();
    delete pAmb;
    return 0;
}
```

子クラスで追加した
sevePeople関数を実行
※親クラスからは実行不可

継承の特徴

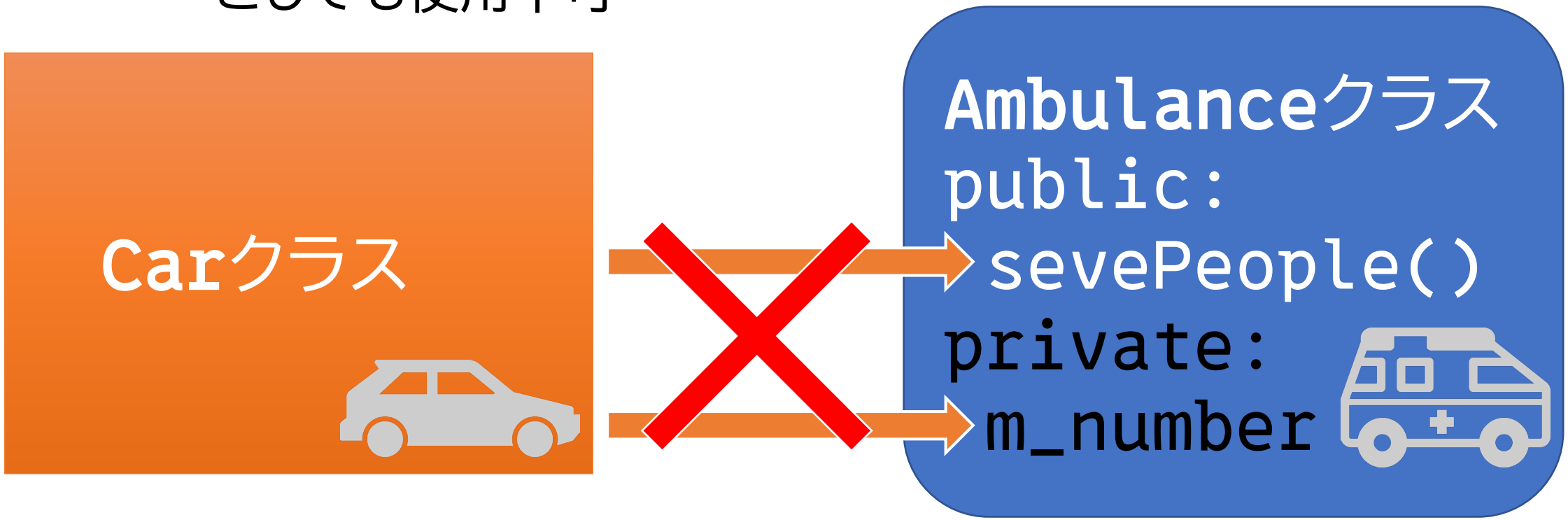
Ambulanceクラスは、Carクラスで
定義されているpublicなメンバ関数
が継承されているので使用可能



privateなメンバは使用不可

継承の特徴

Carクラスからは、Ambulanceクラスで
定義されているメンバはpublicであった
としても使用不可



親クラスからは子クラスの内容は見えない！

継承の特徴

- **親クラス**
  子クラスでも使用する必要最低限のメンバを定義しておき、子クラスで使用しないものは定義しない

- **子クラス**
  継承したメンバ以外に、子クラスで必要なメンバを追加定義して親クラスから機能を拡張していく

継承の特徴

- コンストラクタの実行順

  子クラスのインスタンスが生成されるときに

  ①親クラスのコンストラクタ実行
  ②子クラスのコンストラクタ実行

  という順番に実行される

  コンストラクタの場合は　親　→　子　の順

継承の特徴

- デストラクタの実行順

  子クラスのインスタンスが消去されるときは

  ①子クラスのデストラクタ実行
  ②親クラスのデストラクタ実行

  という順番に実行される

  デストラクタの場合は　子　→　親　の順

継承

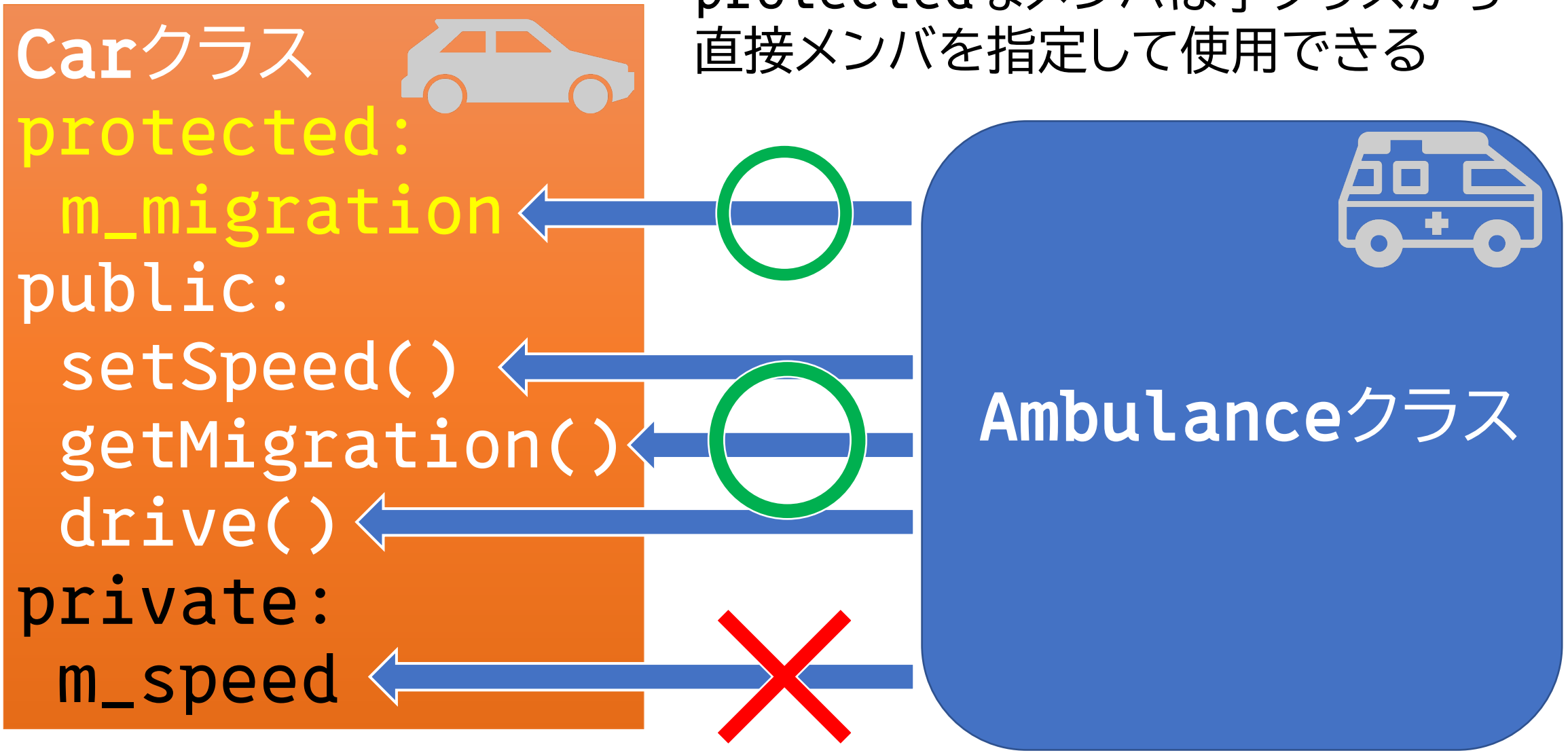
- アクセス指定子 **protected**

- public,private以外の3つめのアクセス指定子

- **protected**に指定されたメンバは子クラスから使用可能
- ただし、クラス外からは使用不可

継承の特徴

protectedなメンバは子クラスから
直接メンバを指定して使用できる



継承

- **多重継承**

  親クラスがひとつの継承を**単一継承**といい、
  親クラスを複数もつ継承を**多重継承**という

  例）

```
class KoClass : public OyaA, public OyaB
```

  ただし、非常に複雑な継承になるため、特別な場合を除いて普段の使用は**単一継承**に留めておく

  他のプログラミング言語では多重継承が禁止されている

ポリモーフィズム

- **オーバーロード(多重定義)**

  クラス内のメンバ関数やコンストラクタは、
  ・引数の型
  ・引数の数量
  を変えることで、同じ関数名であっても、多重定義することができる仕組みがある

  これをオーバーロードと呼ぶ

ポリモーフィズム(オーバーロード)

- 教科書P192~193 **Sample503**

- C++作業フォルダ内に**Sample503**フォルダを作成
```
mkdir Sample503
cd Sample503
```

- calc.h, calc.cpp, main.cppを作成
```
copy nul calc.h
copy nul calc.cpp
copy nul main.cpp
```

ポリモーフィズム(オーバーロード)

- calc.h (Sample503)

```
#pragma once
class Calc {
private:
   int m_a, m_b;
public:
   Calc();
   Calc(int a, int b);
   int add();
   int add(int a, int b);
   void setValue(int a, int b);
   int getA();
   int getB();
};
```

オーバーロードされた関数Calc, add
関数名は同じだが、引数によって呼び出される処理内容が異なる

ポリモーフィ

- calc.cpp (

コンストラクタのオーバーロード
コンストラクタをオーバーロードしたときは
必ず引数のないデフォルトコンストラクタの
定義を行う必要がある！

```
#include "calc.h"
Calc::Calc() : m_a(0), m_b(0) {}
Calc::Calc(int a, int b) : m_a(a), m_b(b) {}
int Calc::add() {
   return m_a + m_b;
}
int Calc::add(int a, int b) {
   return a + b;
}
void Calc::setValue(int a, int b) {
   m_a = a;
   m_b = b;
}
```

(続き)

```
int Calc::getA() {
   return m_a;
}
int Calc::getB() {
   return m_b;
}
```

ポリモーフィズム(オーバーロード)

- main.cpp (SampleF03)

```
#include "calc.h"
#include <iostream>
using namespace std;
int main() {
    Calc* pC1, * pC2;
    pC1 = new Calc();
    pC2 = new Calc(1, 2);
    cout << 3 << "+" << 4 << "=" << pC1->add(3, 4) << endl;
    co
    delete pC1;
    delete pC2;
    return 0;
}
```

pC1: 引数なしのインスタンス
メンバ m_a と m_b は 0

pC2: 引数ありのインスタンス
メンバ m_a は 第一引数
メンバ m_b は 第二引数

```
Calc::Calc() : m_a(0), m_b(0) {}
Calc::Calc(int a, int b) : m_a(a), m_b(b) {}
```

ポリモーフィズム（オー

pC1->add(3,4):
add関数に与えた引数同士を足し算

pC2->add():
メンバ m_a と m_b はインスタンス
生成時に値をすでに代入済みなので、
add関数は m_a + m_b をするだけ

```
int Calc::add() {
   return m_a + m_b;
}
int Calc::add(int a, int b) {
   return a + b;
}
```

```
   pC2 = new Calc(1, 2);
   cout << 3 << "+" << 4 << "=" << pC1->add(3, 4) << endl;
   cout << pC2->getA() << “+” << pC2->getB()
      << "=“ << pC2->add() << endl;
   delete pC1;
   delete pC2;
   return 0;
}
```

ポリモーフィズム

- **オーバーライド(再定義)**

  親クラスと同じメンバを子クラスでも宣言した場合
  子クラスで定義した内容に上書きされる

  これを**オーバーライド**と呼ぶ

```
class Oya {
  void func();
}
```

```
class Ko: public Oya {
  void func();
}
```

こちらを優先して実行

ポリモーフィズム

- **仮想関数**

  教科書P201~203 **Sample506**
- C++作業フォルダ内に**Sample506**フォルダを作成

```
mkdir Sample506
cd Sample506
```

- bird.h, bird.cpp, crow.h, crow.cpp
  chicken.h, chicken.cpp, main.cppを作成

```
copy nul bird.h
copy nul bird.cpp
        （以下略）
```

ポリモーフィズム

- コンパイルの仕方

コマンドプロンプトで次のコマンドを入力する

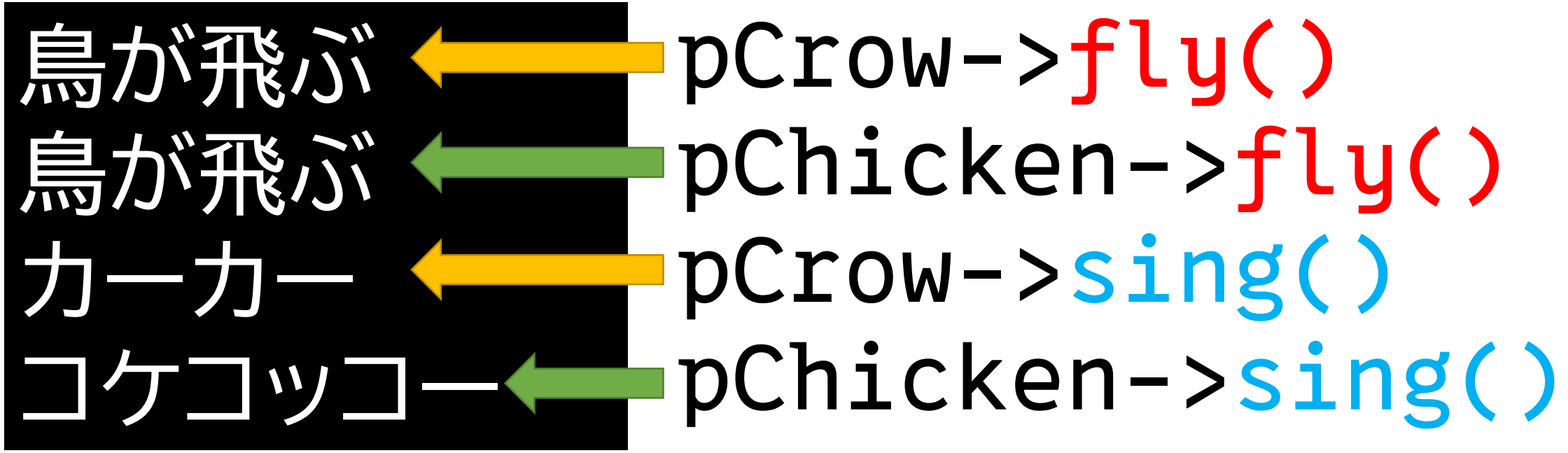
```
cl /EHsc main.cpp bird.cpp
                crow.cpp chicken.cpp
```

※すべてのcppファイルを列挙してください

成功したら、main.exeを実行して結果を確認

ポリモーフィズム

- 実行結果



sing()はオーバーライドできているが、
fly()はオーバーライドできていない

ポリモーフィズム

- **virtual**修飾子
  - sing()がオーバーライドできたのは親クラスのBirdクラスのsing()に**virtual**がついているから！
  - virtualがついた関数は**仮想関数**となり子クラスの同名の関数の方が実行される

ポリモーフィズム

- **純粋仮想関数と抽象クラス**

  教科書P208~209 **Sample507**

- C++作業フォルダ内に**Sample507**フォルダを作成

```
cd ..
robocopy Sample506 Sample507
cd Sample507
```

- bird.h, bird.cpp, main.cppを変更

ポリモーフィズム

- bird.h (Sample507)

```
#pragma once
#include<iostream>
using namespace std;

class Bird {
public:
   virtual void sing() = 0;
   void fly();
};
```

sing() = 0 とすることで、
純粋仮想関数となる

Birdクラスは抽象クラスと
なり、インスタンス化ができなくなる

ポリモーフィズム

- bird.cpp (Sample507)

```
#include "bird.h"

void Bird::sing() {
   cout << "鳥が鳴く" << endl;
}
void Bird::fly() {
   cout << "鳥が飛ぶ" << endl;
}
```

純粋仮想関数となった関数の定義は不要

ポリモーフィズム

- main.cpp (Sample507)

```
int main()
{
  Bird* pCrow{}, * pChicken{}, * pBird{};
  pCrow = new Crow();
  pChicken = new Chicken();
  pBird = new Bird();
  pCrow->fly();
  pChicken->fly();
  pCrow->sing();
  pChicken->sing();
  delete pCrow;
  delete pChicken;
```

コンパイルエラー

抽象クラスのインスタンスを
生成することはできない！

ポリモーフィズム

- 仮想デストラクタ

  親クラスのデストラクタに`virtual`を付けると…

  子クラスのデストラクタ実行後に
  親クラスのデストラクタが実行されるようになる！

  子クラスのデストラクタで終了時処理に処理漏れがあった際、親クラスのデストラクタでカバーができる利点がある

ポリモーフィズム

- **仮想デストラクタ**

親クラス

Fatherクラス

Father()
~Father()

継承

子クラス

Sonクラス

Son()
~Son()

インスタンス消去時…

~Son()のみ
実行される

親クラス

Motherクラス

Mother()
virtual ~Mother()

継承

子クラス

Daughterクラス

Daughter()
~Daughter ()

~Daughter()
のあとに
~Mother()
が実行される