C++講座最終日

夜食
@yasyoku_

前回までのお話

- クラスとは
  - ヘッダーファイル(CLASSNAME.h)とソースファイル(CLASSNAME.cpp)で構成する感じ。

- 今日の話はクラスをメインで行います。
  とりあえず次のクラスを構成してください。

Creature.h

```
class Creature {
private:
    char* name;
    int hp;
    int mp;
public:
    Creature();
    Creature(char* name);
    ~Creature();

    virtual void print();

    void set_name(char* name);
    char* get_name();
};
```

Creature.cpp

```
#include <iostream>
#include "Creature.h"
 using namespace std;

Creature::Creature(){
    this->name = "null";
}

Creature::Creature(char* name){
    this->name = name;
}
 Creature::~Creature(){}

void Creature::print(){
    cout << "このCreatureクラスのnameは" << name << "です¥n";
}

void Creature::set_name(char* name){
    this->name = name;
}
char* Creature::get_name(){
    return name;
}
```

virtual は後々説明します。

とりあえず動かして試してみる

- Main文はこんな感じでprintメソッドの動作を確認してみる。

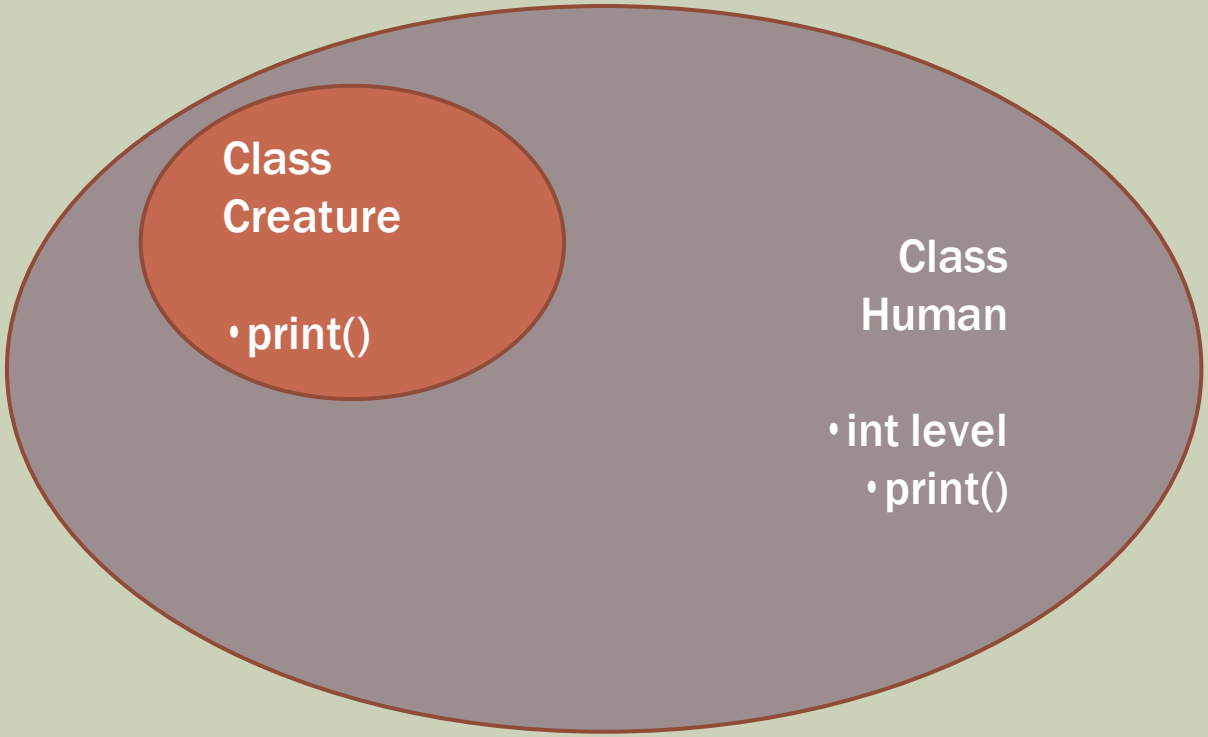
```
#include <iostream>
#include "Creature.h"

int main(){
    char namae[] = "Test";
    Creature test_Creature = Creature(namae);

    test_Creature.print();
}
```

```
このCreatureクラスのnameはTestです
続行するには何かキーを押してください . . .
```

- こんな感じで動けば問題なし。

今日のお話

- クラスを拡張してみる。
  - 生物クラスから人間クラスを作ってみる。
  - RPG的なかんじでlevelも追加してみる。



とりあえずソースコード

Human.h

```
#include "Creature.h"

class Human : public Creature{
private:
    int level;
public:
    Human();
    Human(char* temp_name);
    ~Human();

    void print();
};
```

Human.cpp

```
#include "Human.h"
#include <iostream>

void Human::print(){
    std::cout << "このHumanクラスのnameは" << get_name() << "だよ¥n";
}

Human::Human(){};

Human::Human(char* temp_name){
    set_name(temp_name);
}

Human::~Human(){};
```

継承するときの書き方(ヘッダー)

- #include “継承もとクラスのヘッダーファイル.h”
  - このとき#include <iostream> と書式が違うことに注意。自分で作成したファイルは” ”を用いります。
    親クラスのデータをインクルードします。

- Class Human : public Creature
  - Class HumanをCreatureのpublicの範囲で作成。
    - public：基底クラスで設定したアクセス修飾子の設定をそのまま引き継ぐ
    - protected：基底クラスでpublicだったものを、protectedにして引き継ぐ。他はそのまま。
    - private：基底クラスのメンバを全てprivateで引き継ぐ。
  - とまあいろいろありますけどpublicだけで十分です。

```
#include "Creature.h"

class Human : public Creature{
private:
    int level;
public:
    Human();
    Human(char* temp_name);
    ~Human();

    void print();
};
```

継承するときの書き方(ソース)

- クラスを継承する時もソースファイルの書き方は同様です。

- 親クラスCreatureにもprint()が存在するのに子クラスHumanにもprint()が存在すると多重定義でエラーになるのでは？

```
#include "Human.h"
#include <iostream>

void Human::print(){
    std::cout << "このHumanクラスのnameは" << get_name() << "だよ¥n";
}

Human::Human(){};

Human::Human(char* temp_name){
    set_name(temp_name);
}

Human::~Human(){};
```

オーバーライド(OVERRIDE)

- 親クラスと子クラスに同じ関数があり、子クラスで再定義した場合子クラスでの定義を子クラスでは優先して使用する。(OverRide)
- 親クラスでvirtualをつけないとOverrideは行われない。

Creature.h

```
class Creature {
private:
    char* name;
    int hp;
    int mp;
public:
    Creature();
    Creature(char* name);
    ~Creature();

    virtual void print();

    void set_name(char* name);
    char* get_name();
};
```

Human.h

```
#include "Creature.h"

class Human : public Creature{
private:
    int level;
public:
    Human();
    Human(char* temp_name);
    ~Human();

    void print();
};
```

試してみる

- これの実行結果を見てみる。

```
#include <iostream>
#include "Human.h"

int main(){
    char namae[] = "TEST";
    Creature test_Creature = Creature();
    Human test_Human = Human(namae);

    test_Creature.print();
    test_Human.print();
}
```

```
このCreatureクラスのnameはnullです
このHumanクラスのnameはTESTだよ
続行するには何かキーを押してください . . .
```

- 表記がしっかり変わってると思います。

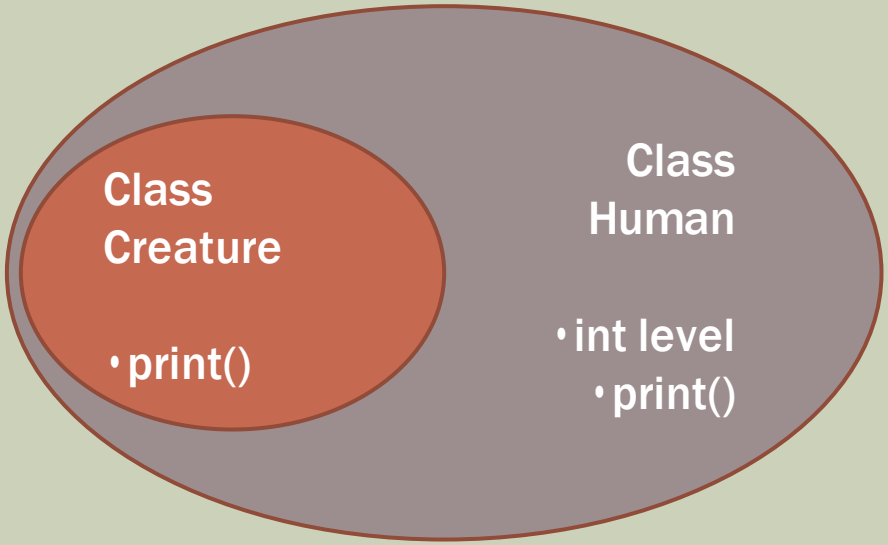
親と子

- 親クラスのCreatureの場合
  - Creature temp_Creature = Creature();
  - temp_Creature.print();
- 子クラスのHumanの場合
  - Human temp_Human = Human ();
  - temp_Human.print();
- こんな感じで同じようなこと何回も書くの馬鹿らしいっすね。

アップキャスト

- ここで、派生クラス(Human)は親クラス(Creature)の全てのメンバを持っていることに注目。

- Creatureクラスの範囲であればHumanクラスの内容も取り扱うことが出来るはず。(Creature ⊆ Humanのため)
- じゃあ親に入れ込んじゃおう



アップキャスト

- Creature *TEST[2];
- TEST[0] = new Creature();
- TEST[1] = new Human();

- こんな風に書くことができます。
  子を親に突っ込みました(アップキャスト)

- アップキャストを行うときは必ずポインターで行って下さい。
  行わない場合はCreatureクラスとして扱われてしまいます。

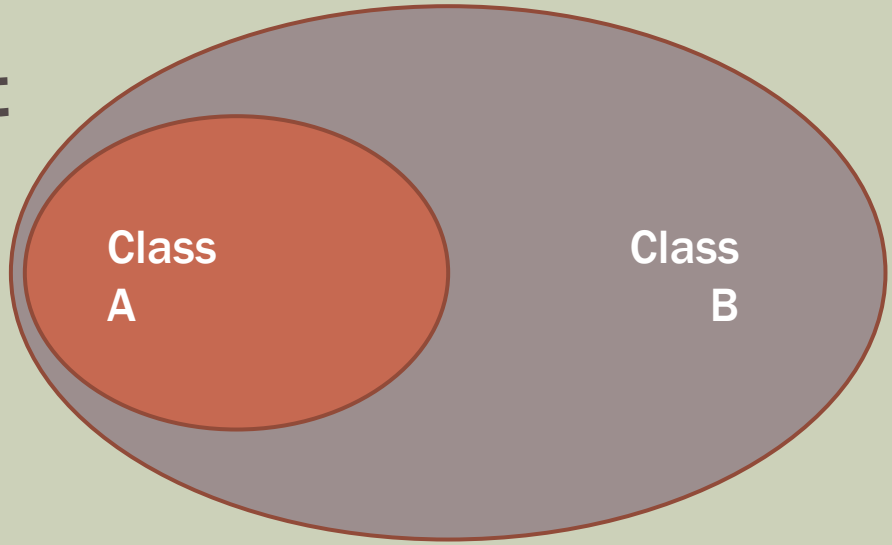
- newしたらdeleteするのも忘れずに。

アップキャストの注意点

- 親クラスをA、子クラスをBとする。

AにA	可能
AにB	可能
BにA	不可能
BにB	可能

- BにAを入れると$\bar{A} \cap B$の部分に対応できない！！



アップキャストとオーバーライド

- アップキャストとオーバーライドを行うことによる利点。
  以下の場合の実行結果が変化します。

```
Creature *TEST[2];
TEST[0] = new Creature();
TEST[1] = new Human();

TEST[0]->print();
TEST[1]->print();
```

```
このCreatureクラスのnameはnullです
このHumanクラスのnameはnullだよ
続行するには何かキーを押してください . . .
```

- こんな感じでオーバーライドした関数が自動的に呼び出される

- ちなみにvirtualを抜いたときの実行結果が次のやつ。

```
このCreatureクラスのnameはnullです
このCreatureクラスのnameはnullです
続行するには何かキーを押してください . . .
```

まとめ

- オーバーライドとアップキャストを行うと自動的に関数を認識してやってくれる！
- オーバーライドする関数はvirtual演算子をつける。
- アップキャストするときはポインターで扱う。

オーバーロード

- 分数の話です。
  - クラスFractionはプライベートメンバ分母、分子を持ってる。
  - printすると分数の形で印字される。

- 目的：num1+num2(分数の足し算)をしたい！！

Fraction.h

```
#include<iostream>

class Fraction{
private:
    int upper;  //上の数
    int downer; //下の数
public:
    Fraction();
    Fraction(int num1, int num2);
    ~Fraction();

    void print();
};
```

main

```
#include"Fraction.h"

int main(){
    Fraction num1 = Fraction(1, 3);
    Fraction num2 = Fraction(1, 2);

    num1.print();
    num2.print();

}
```

```
1/3
1/2
続行するには何かキーを押してください . . .
```

オーバーロード

- 普通にnum1 + num2ってやろうとするとコンパイルできません。
  (どうやって足し算すればいいのかコンパイラが判ってない)

- じゃあこうすりゃいいんだろ！！

  - Fraction num3 = Fraction(num1.upper*num2.downer + num1.downer*num2.upper, num1.downer*num2.downer);

- よく読めば理解できますが……。
  毎回こんなこと書くの**非効率**です。

オーバーロード

- ここで**演算子オーバーロード**というものが活躍します。

```
Fraction Fraction::operator+(Fraction obj){
    return Fraction(this->upper*obj.downer + this->downer*obj.upper, this->downer*obj.downer);
}
```

- こんな感じのコードを書き加えると、
  - Fraction num3 = num1+num2;
- が実行できるようになります。演算子の定義を変えたというのが演算子オーバーロードになります。

```
Fraction num3 = num1 + num2;
num3.print();
```

```
1/3
1/2
5/6
続行するには何かキーを押してください . . .
```

オーバーロード

- operatorキーワードによって +,-,*,/,^,...etc 様々な演算子を定義できます。
- また、演算子オーバーロードの定義文では左辺は暗黙的にthisポインタで渡されます。(num1 + num2)

```
Fraction Fraction::operator+(Fraction obj){
    return Fraction(this->upper*obj.downer + this->downer*obj.upper, this->downer*obj.downer);
}
```

  - javaにはこの機能は「混乱を招く」という理由な感じで実装されてません。

試してみる

- クラスFractionを作成する。
  - メンバーに分母、分子をもたせる。
  - 引数に分母分子を入れるコンストラクターを作成する。
  - 印字用の関数を作成する。

- operator –　を作成してみましょう。

てんぷれ

```
int Max( int a, int b){
    return ( a > b ) ? a : b;
}

float Max( float a, float b){
    return ( a > b ) ? a : b;
}

double Max( double a, double b){
    return ( a > b ) ? a : b;
}
```

- やってることはどれも一緒

テンプレ

- 理想
  - TYPE Max(TYPE a, TYPE b){
  - 　return (a > b) ? a : b;
  - }
- 実装
  - template <typename TYPE> TYPE Max(TYPE a, TYPE b){
  - 　return (a > b) ? a : b;
  - }
- こんな感じで勝手に型を認識してくれる

```
int num5 = 5;
int num7 = 7;
std::cout << Max(num5, num7) << std::endl;
```

```
7
続行するには何かキーを押してください . . .
```

TEMPLATE

```
template <typename TYPE> TYPE Max(TYPE a, TYPE b){
    return (a > b) ? a : b;
}
```

- ならFractionクラスでも適応できるのか？
  - できますが今の状態では出来ません。出来るように直してみてください。

ちなみに

- クラスをtemplateで扱えるので、できれば**参照渡し**を行いたい。

```
template <typename TYPE> TYPE& Max(TYPE& a, TYPE& b){
    return (a > b) ? a : b;
}
```

- こうするだけ。
- <typename TYPE>のところは飽くまで**命名**なので**&**をつけないこと。

- <typename TYPE>を使うと**クラスのメンバ**など様々なことにも利用できます。

FRIEND

- クラスFractionに整数(int)を引いてみよう。
  - Fraction operator- (int int_value, fraction fraction_obj)
- こんな感じで、左辺値が同じクラスで無い場合は指定してあげることが出来ます。ただしメンバ関数ではなく**グローバル関数**として作成してください。
  - Fraction operator- (int int_value, Fraction fraction_obj){
    - return Fraction(
    - int_value*fraction_obj.downer - fraction_obj.downer,
    - fraction_obj.downer
    -);
  - };

FRIEND

- 先ほどのコードだと反応しません。
- **普通は**そのクラスのメンバー変数はprivate であることが多い。
- privateだとクラスの外からは覗けない。

```
#include<iostream>

class Fraction{
private:
    int upper;  //上の数
    int downer; //下の数
public:
    Fraction();
    Fraction(int num1, int num2);
    ~Fraction();

    void print();
};
```

```
Fraction operator- (int int_value, Fraction fraction_obj){
    return Fraction(int_value*fraction_obj.downer - fraction_obj.upper, fraction_obj.downer);
};
```

FRIEND

```
Fraction operator- (int int_value, Fraction fraction_obj){
    return Fraction(int_value*fraction_obj.downer - fraction_obj.upper, fraction_obj.downer);
};
```

- このような時に、この関数はFractionの一員ですよー、ということを示すfriendを付け加える。
- friendはグローバル関数を擬似的にメンバ関数にする。
  そのため、宣言の位置をグローバルではなくFractionのメンバとして移動する。

```
class Fraction{
private:
    int upper;  //上の数
    int downer; //下の数
public:
    Fraction();
    Fraction(int num1, int num2);
    ~Fraction();

    void print();

    Fraction operator + (Fraction obj);
    bool operator > (Fraction obj);
    friend Fraction operator- (int int_value, Fraction fraction_obj);
};
```