# 応用プログラム言語II / 演習II
# #07
## C++で学ぶオブジェクト指向プログラミング（その4）

情報システム学科
坂本政祐
sakapon@sit.ac.jp



## はじめに（#04復習）

復習

- OOPには三大要素というものがあります。
  - クラス（済）
  - ポリモーフィズム（済）
  - 継承（済）
- ポリモーフィズムと継承については，より便利に使える仕組みがあるので紹介します。



# 準備1:
# インスタンスのもう1つの作り方



## 準備1のはじめに

- ポリモーフィズムをさらに便利に使う方法があるわけですが，それを用いるためには，先に「インスタンスの動的な生成」を知っておかないといけません。
- ので，まずは準備としてその生成法を学びましょう。



## これまでのインスタンスの作り方

- これまでに出てきたインスタンスの作り方:
  - クラス名 インスタンス名
    または
  - クラス名 インスタンス名(コンストラクタに渡す引数)

リスト7-1
```
class Teki
{
  /* 省略 */
};

int main(void)
{
  Teki boss;
  ...
  boss.ugokasu();
  ...
}
```



## これまでのインスタンスの作り方

- これは，静的なインスタンスの作り方。
  - Cでも，静的な変数の作り方と動的な変数の作り方があった。
    - 静的:
```
int num;
num = 365;
```
    - 動的:
```
int *num;
num = (int *)malloc(1 * sizeof(int));
*num = 365;
```
  - ちなみに，静的と動的の違いは:
    - 静的: コンパイル時に，変数の有無や個数が決まっている場合。
    - 動的: コンパイル時には変数の有無や個数がわからないので，実行時にそれらが決まる場合。



## C++での動的なインスタンスの作り方

- C++でも，new演算子を使って動的にインスタンスを作れる。

リスト7-2
```
class Teki
{
  /* 省略 */
};

int main(void)
{
  Teki *boss;
  boss = new Teki;
  ...
  boss->ugokasu();
  ...
}
```



## new演算子で動的インスタンスを作る際の注意点

- 動的なインスタンスの生成での注意点:
  - new演算子の評価結果は「動的に確保できたアドレス」なので，インスタンス用の変数もポインタ変数にしなければならない。
  - 「Teki boss;」と「Teki *boss; boss = new Teki」のイメージとしての違い:
    - 前者は，インスタンスそのものに直接 boss という名前をつけたイメージ。
    - 後者は，new Teki で new 演算子にインスタンスを生成してもらって，その時点では名無しのインスタンスがふらふら漂っている状態だが，それを指し示す名前として boss という名前を用意したことで，ふらふら状態ではなく間接的にではあるが boss という名前で使えるようにしたイメージ。

## new演算子で動的インスタンスを作る際の注意点

applied-progII #07

- 動的なインスタンスの生成での注意点（続き）:
  - ポインタの場合は，クラスのメンバ変数やメンバ関数には「.」ではなく「->」でアクセスしなければならない。
    - この辺はCにおける構造体のそれとそっくり。
  - 宣言と初期化は以下のように同時にやっても構わないが，ここではわかりやすさのために2行に分けて書く方法で統一する。

```
Teki *boss = new Teki;
```

  - コンストラクタに引数を渡したい場合は，「new クラス名」の直後に「(引数)」をくっつければよい。

```
Teki *boss;
boss = new Teki("boss.bmp");
```

  - new したものは本当は自分で delete しなければいけないのだが，ここではその話は置いておく。

9

## 準備2:
## オーバーライドと仮想関数

10

## 準備2のはじめに

applied-progII #07

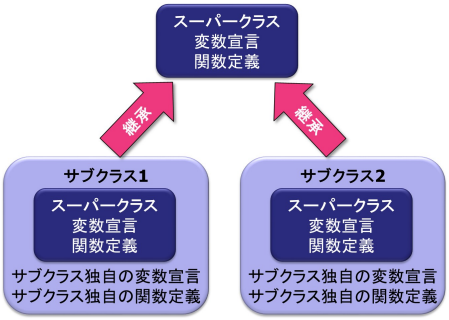
- 今から「オーバーライド」と「仮想関数」について説明します。が，
  - C++という言語そのものを学習する場合は重要な概念ではあるものの，
  - C++が，OOPを学習するための手段でしかない場合（この講義がそう）は，それほど重要でもない。
  - ポリモーフィズムの感動的使い方のためには先にこれ言っておかないと説明できないので仕方なく説明しますが，「そういうものか」的に聞いてくれれば良いです。

11

## 継承関係の図

applied-progII #07

- 前回，こういう図がありました。




12

## 関数のオーバーライド

applied-progII #07

- サブクラス側では:
  - スーパークラスの変数や関数をそのまま取り込める。
  - サブクラス独自の変数や関数を新たに定義できる。

  ということだった。
- これにさらに加えて，スーパークラスで既に定義されている関数を，サブクラスで同じ名前で上書きしてしまうこともできる。
- これを，関数のオーバーライドという。
  - オーバーロードとはまた別です。なんでこんなややこしい名前にしたのか…。

13

## 関数のオーバーライド

applied-progII #07

- オーバーライドには，そのための特別な文法はなく，スーパークラスにあった関数を同名で定義しなおせばそれはオーバーライドしたことになる。
- このとき，オーバーライドされる側（スーパークラス側）の関数は，
  - オーバーライドされても構わないんだけど，その関数名がスーパークラスにも存在することが大事なために定義しておく仮想関数と，
  - オーバーライドされたとしても，スーパークラスで定義されていること自体の意味は失わない，仮想じゃない関数，
  - とに分類できる。よくわからないですよね…今回は仮想関数しか使いません。

14

## 仮想関数

applied-progII #07

- 仮想関数（と自分で決めた関数）には，その定義の際に定義の先頭に virtual をくっつけます。

```
class Kyara
{
public:
  virtual void ugokasu(void)
  {
    // どうせオーバーライドされるので中身は空っぽ
  }
};

class Teki : public Kyara
{
public:
  void ugokasu(void) // これがオーバーライド
  {
    // Tekiクラス独自の動き
  }
};
```

Point

15

## ポリモーフィズムの真髄

16

復習

## 前回，こういうスライドがありました

applied-progII #07

- ポリモーフィズムのところで書いたように，継承の仕組みを利用するとポリモーフィズムがさらに活きてきます。
- なんてスマートな書き方なんだ！とウケること請け合いですが，それはまた次回。
- これをここで明らかにします。

17

## サブクラスのインスタンスはスーパークラスのインスタンスのように振る舞える

applied-progII #07

- 動的に生成したインスタンスは，そのクラスのスーパークラスのインスタンスのようにも扱うことができる。（静的なインスタンスはダメ）

リスト7-3

```
class Kyara
{
  /* 省略 */
};

class Teki : public Kyara
{
  /* 省略 */
};

int main(void)
{
  Kyara *kyara1;

  kyara1 = new Teki;
}
```

ここで宣言しているのは Kyara クラスへのポインタなのに...

こっちでは Teki クラスのインスタンスを new した結果を代入している。

18

## サブクラスのインスタンスはスーパークラスのインスタンスのように振る舞える

applied-progII #07

- なので，スーパークラス用の std::vector の1要素にもなれる。

リスト7-4

```
#include <vector>

class Kyara
{
  /* 省略 */
};

class Teki : public Kyara
{
  /* 省略 */
};

int main(void)
{
  std::vector<Kyara *> kyaras;

  Teki *boss;
  boss = new Teki;

  kyaras.push_back(boss);
}
```

Point

Point

- ちなみに，new したものを格納するので，

```
std::vector<Kyara> kyaras;
```

ではなく，

```
std::vector<Kyara *> kyaras;
```

である。

19

## ポリモーフィズムの真髄3秒前

applied-progII #07

- ということは，ポリモーフィズムの説明のときに書いていた，以下のような zako と boss と jiki をそれぞれ ugokasu() ような書き方は，

```
zako.ugokasu();
boss.ugokasu();
jiki.ugokasu();
```

- 以下のようにそれぞれを new で作ってひとつの std::vector に入れておきさえすれば...

```
std::vector<Kyara *> kyaras;

Teki *zako;
zako = new Teki;
kyaras.push_back(zako);

Teki *boss;
boss = new Teki;
kyaras.push_back(boss);

Jiki *jiki;
jiki = new Jiki;
kyaras.push_back(jiki);
```

20

## ポリモーフィズムの真髄

applied-progII #07

- この std::vector の中身ひとつひとつに，ugokasu() しちゃえば良いのです!!

```
for(int i=0; i<kyaras.size(); i++){
  kyaras[i]->ugokasu();
}
```

- すごく感動的にスマートな書き方ですね!!　本当だったら，キャラが100個画面上にいたら，100行◯◯.ugokasu() と書かねばいけないところを。
- なお，こう書けるためには，スーパークラス側で ugokasu() 関数が仮想関数として定義されてある必要があります。

21

## うごくサンプル

applied-progII #07

- というわけで，動くサンプルとしては以下のようになる。
  - 各 ugokasu() 関数には，便宜的に画面出力をいれてある。
  - 他のメンバ関数は省略してある。

リスト7-5

```
#include <iostream>
#include <vector>

// キャラクタ用汎用クラス
class Kyara
{
public:
  virtual void ugokasu(void)
  {
    std::cout << "キャラ動く。" << std::endl;
  }
};

（続く）
```

22

## うごくサンプル

applied-progII #07

リスト7-5（続き）

```
// 敵クラス（class Kyara を継承）
class Teki : public Kyara
{
public:
  void ugokasu(void)
  {
    std::cout << "敵動く。" << std::endl;
  }
};

// 自機クラス（class Kyara を継承）
class Jiki : public Kyara
{
public:
  void ugokasu(void)
  {
    std::cout << "自機動く。" << std::endl;
  }
};

（続く）
```

23

## うごくサンプル

applied-progII #07

リスト7-5（続き）

```
int main(void)
{
  // あらゆるキャラ（へのポインタ）を格納するコンテナ
  std::vector<Kyara *> kyaras;

  // いろんな敵も自機もそのコンテナに入れておき...
  Teki *boss;
  boss = new Teki;
  kyaras.push_back(boss);

  Teki *zako;
  zako = new Teki;
  kyaras.push_back(zako);

  Jiki *jiki;
  jiki = new Jiki;
  kyaras.push_back(jiki);

  // ポリモーフィズムで全キャラ一気に動かす！
  for(int i=0; i<kyaras.size(); i++){
    kyaras[i]->ugokasu();
  }
}
```

24

# C++で学ぶ オブジェクト指向プログラミング まとめ



## OOP with C++ まとめ



- 4回に渡って，オブジェクト指向プログラミング(OOP)について学んできました。
  - 言語はC++を用いました。
- が，正直なところそれほど劇的に便利だという実感はわいていないのではないかと思います。
- それでも，現時点ではそれで良いと思います。
- OOPは，クラスを設計するセンスも割と必要ですし，一度設計したクラスを解体しては再構築したり，一部をスーパークラスに移したり，といった試行錯誤もよく起こりますが，



## OOP with C++ まとめ



- そういった試行錯誤は，
  - 比較的大規模なプログラムや，
  - 小規模でも何度も再利用するプログラム

  においてよく本領を発揮しますから，講義のような，プログラミングのほんの一面を切り出して学ぶようなやり方では，感動的な実感は得にくいのは仕方ないのかもしれません。
- ただ，OOPという引き出しを自分の中に持ってあるのと無いのでは将来随分違うはずです。



## OOP with C++ まとめ



- Cで書き始めたは良いが，なんか猛烈に面倒くさいデータ構造や猛烈に面倒くさいアルゴリズムになりそうだぞ，と思ったときに，「そういや，OOPというなんかプログラムの無駄を省いて整理整頓するための仕組みを昔習ったな」という感じでその引き出しを開けてくれれば良いなと思います。



## OOP with C++ まとめ



- また，フリーなプログラミングライブラリも沢山配布されています。
- そういったものをこれから利用するとき，「C++用だから」というだけで，とても優れた使い勝手のよいライブラリなのにそれを避けるというもったいないこともしなくて済むでしょう。



## OOP with C++ まとめ



- 例えば，yaml-cpp という，YAMLフォーマットファイルをパースするためのC++用ライブラリがあります。これを使ったコード例は以下のようになります。
- 4週間前なら，これって何？ 状態だったと思います。

```
#include <ifstream>
#include "yaml-cpp/yaml.h"

int main(void)
{
  std::ifstream fin("test.yaml");
  YAML::Parser parser(fin);

  YAML::Node doc;
  while(parser.GetNextDocument(doc)) {
    // ...
  }

  return 0;
}
```

- でも今なら，
  - parser はYAML::Parserクラスのインスタンスで，コンストラクタにfinを渡しているんだな，
  - parser に対して，GetNextDocument()関数を呼んでいるんだな，というのがわかりますよね!



## OOP with C++ まとめ



- また，卒業研究などでC++やJavaで書くことになったとき，「オブジェクト指向? 何それ?」状態にならずにも済むでしょう。
- というわけで，これからも Happy OOPing!



## 今日の課題



- 課題1: 先週までに自分が作ってきたクラスで，ポリモーフィズムを取り入れて実装してください。

## 今週の落穂拾い

applied-progII #07

- C++ではnew演算子が導入されたせいで，new という変数名が使えなくなった。「1回ループを回すたびに，現在の状態を new という変数に入れて，ひとつ前の状態を old という変数に入れる」というのは割と良くやるので，結構この罠には良くひっかかります（笑）。

33

## 今週の落穂拾い

applied-progII #07

- また，C++には他にも「テンプレート」とか「演算子のオーバーロード」とか「例外」とかの魔窟…じゃなかった，モダンなプログラミング機構がいろいろ用意されています。興味があったらぜひ極めてみてください。

34