4. C++ (4) 「継承とポリモーフィズム」

♣ C++ における「継承」がどのようなもので, どんなメリットがあるか説明する. また, 継承を用いたプログラミングの例や「仮想関数」についても説明する.

4.1 では, まず継承とは何か簡単に説明する.

4.2 では, 継承の応用例の 1 つとして, 既存のクラスを拡張したクラスを作成する方法を紹介する.

4.3 では, 別の応用例として, 既存のクラスに機能を追加する方法について演習を行う.

4.4 と 4.5 では, 「ポリモーフィズム」と呼ばれる継承の応用を紹介する. 図形の描画の例を用いて, この手法を説明する.

4.1 継承とは

継承 (inheritance) はオブジェクト指向の重要な概念で, 種々の観点からの説明ができるが, C++ のプログラミングに限定して簡単に言えば, 「既存のクラスを元にして新しいクラスを作る仕組み」と考えることができる.

例えば, メンバー 100 個, メンバ関数 50 個からなる「パソコン」のというクラスを設計した後に, 「ノートパソコン」のクラスを設計しなければならなくなったとする. PCMCIA カードやバッテリーのデータがあるので「パソコン」にメンバーとメンバ関数を追加してクラスを作ることになる. その際に, 「パソコン」とほぼ同じ「ノートパソコン」のクラスを一から定義するのは, とても無駄な作業と感じられるだろう. コピーして必要な部分だけ書き換えれば手間の問題は解決するが, 同じようなコードが何箇所にも存在すると, データ構造の変更があった場合の修正が難しくなり, プログラムの保守という観点から好ましくない.

「継承」は, あるクラスから別のクラスをつくり出す仕組みで, この例で言えば, 「『ノートパソコン』は『パソコン』の性質を『継承』する」と宣言し, 追加分の記述だけ行えば新しい「ノートパソコン」のクラスができ上がる, というものである.

これによって, 手間が省けるのはもちろんであるが, データ構造やプログラムが整理された形で書け, 大規模なプログラムの開発が容易になるというメリットがある.

4.2 例 1—少し異なる新しいクラスの作成

4.2.1 元になるクラス (基底クラス)

まず元になるクラスとして, 次の car を考える.

1. メンバーとしては車に関するいくつかのデータを記憶している.

[List 4.1]

```
 1:  class car {
 2:     private:
 3:        std::string m_model;     // 名前
 4:        std::string m_maker;     // メーカー
 5:        int m_displacement;     // 排気量
 6:        int m_ps;     // 最高出力
 7:        int m_weight;     // 車重
 8:     public:
 9:        メンバー関数 [List 4.2]
10:  };
```

コーディング規約

[List 4.1] では, メンバーの変数名がすべて m_ で始まっている. グローバル変数は g_ で始める, ++ は用いない等, プログラムの保守性向上のために, 会社や部署毎にローカルな取り決めをすることがあり, これらを「コーディング規約」と呼ぶ. 「はぁ!?」と思うようなコーディング規約に従わされることもあるらしい.

2. メンバー関数は, いくつかのデータを算出する.

[List 4.2]

```
 1:  car() {}
 2:
 3:  car(const std::string& md, const std::string& mk, int dp, int p, int w):
 4:     m_model(md), m_maker(mk), m_displacement(dp), m_ps(p), m_weight(w) {}
 5:
 6:  ~car() {}
 7:
 8:  std::string model() const {return m_model;}    // 名前を返す
 9:
10:  int no() const {  // 3 ナンバーか 5 ナンバーか
11:     if (m_displacement<2000) return 5;
12:     else                     return 3;
13:  }
14:
15:  double pwratio() const { // パワー・ウェイト・レシオ
16:     return (double) m_weight/(double) m_ps;
17:  }
18:
19:  int tax() const {  // 自動車税
20:     if      (m_displacement<=1000) return  29500;
21:     else if (m_displacement<=1500) return  34500;
22:     else if (m_displacement<=2000) return  39500;
23:     else if (m_displacement<=2500) return  45000;
24:     else if (m_displacement<=3000) return  51000;
25:     else if (m_displacement<=3500) return  58000;
26:     else if (m_displacement<=4000) return  66500;
27:     else if (m_displacement<=4500) return  76500;
28:     else if (m_displacement<=6000) return  88000;
29:     else                           return 110000;
30:  }
```

3. 次のようなメインでデータを与え表示させる.

[List 4.3]

```
 1:  int main(void) {
 2:
 3:      car a = car("Skyline", "Nissan", 3498, 272, 1500);
 4:      car b = car("Civic", "Honda", 1998, 215, 1190);
 5:
 6:      std::cout << a.model() << "  "
 7:                << a.no() << "  "
 8:                << a.pwratio() << "  "
 9:                << a.tax() << std::endl;
10:
11:      std::cout << b.model() << "  "
12:                << b.no() << "  "
13:                << b.pwratio() << "  "
14:                << b.tax() << std::endl;
15:
16:      return 0;
17:  }
```

これを実行すると,

```
Skyline  3  5.51471  58000
Civic  5  5.53488  39500
```

などと表示される.

4.2.2 新しいクラス (派生クラス)

car を元に, 新たに「ハイブリッドカー」のクラス hybrid_car を作ってみよう. car に 対する変更は,

1. モータの出力を示す int m_motor_ps が追加される.
2. pwratio() の計算が変わる (最高出力にモーターの馬力を加算)
3. tax() の計算が変わる (税金が安くなる)

の 3 点で, 後は元の car と同じとする. そのようなクラスは, 継承を用いると次のように作れる.

1. クラスの定義において, 「car を継承する」ことを宣言する.
   クラス宣言の部分「class hybrid_car」を「class hybrid_car : car」と書く.
   これにより, hybrid_car はデフォルトで car の性質をそっくり受け継ぐことができる.

   - このとき, 元になるクラスを**基底クラス (base class)**, 新たに作られるクラスを**派生クラス (derived class)** という. 先の例では, car が基底クラスで hybrid_car が派生クラスである.

     ※ 基底クラス, 派生クラスをそれぞれスーパークラス (superclass), サブクラス (subclass) と呼ぶこともある.

[List 4.4]

```
1:  class hybrid_car : public car {   // public の意味は下記参照
2:     private:
3:        int m_motor_ps;
4:     public:
5:        メンバー関数 [List 4.5]
6:  };
```

2. データは, **新たに追加した分だけ宣言**すればよい.

   [List 4.4] のように, m_motor_ps だけ宣言すれば, もともと car にあった 5 つのメンバー に加えて m_motor_ps を持つクラスができ上がる.

   - 1 行目のキーワード public は, 基底クラス car のメンバーを public としてアクセスすることを宣言している. もし, これらのメンバーを外部からのアクセスさせたくない場合は private と書く.

   ※ この点に関しては, すぐ後でコメントする.

3. メンバー関数も, **新たに追加した分と変更のある分**だけ記述すればよい.

   基底クラスと同じ名前のメンバー関数を定義すると, 新しいものが優先される. (「overwrite (上書き) される」という.)

[List 4.5]

```
 1:     hybrid_car() {}
 2:
 3:     hybrid_car(const std::string& md, const std::string& mk,
 4:        int dp, int p, int w, int mps):
 5:        car(md,mk,dp,p,w), m_motor_ps(mps) {}
 6:
 7:     ~hybrid_car() {}
 8:
 9:     double pwratio() const { // 最高出力にモーターの馬力を加算
10:        return (double) m_weight/(double)(m_ps+m_motor_ps);
11:     }
12:
13:     int tax() const {return 0;}  // 「税金がタダ」 (後で書き換える)
```

4. 基底クラス car を 1 点だけ書き換える必要がある. (少しだけややこしい)

   - 実は, 基底クラスでメンバーやメンバー関数を private と宣言すると, 継承クラスからもアクセスができなくなってしまう (継承の際に public と宣言してもダメ).
   - 基底クラスのあるメンバについて, 継承クラスにはアクセスを許可したいが, 他のクラスからのアクセスは許可したくないしたいというワガママ (ではなく, 比較的普通の) 状況に対応する場合には, **基底クラスの宣言を private から protected に変更**する.

[List 4.6]

```
 1:  class car {
 2:     protected:
 3:        std::string m_model;     // 名前
 4:        std::string m_maker;     // メーカー
 5:        int m_displacement;     // 排気量
 6:        int m_ps;     // 最高出力
 7:        int m_weight;     // 車重
 8:     public:
 9:        メンバー関数 [List 4.2]
10:  }
```

5. 新しいメインを書いて実行してみよう. [List 4.3] に下線部を追加する.

[List 4.7]

```
 1:  int main(void) {
 2:
 3:      car a = car("Skyline", "Nissan", 3498, 272, 1500);
 4:      car b = car("Civic", "Honda", 1998, 215, 1190);
 5:      hybrid_car h = hybrid_car("Prius", "Toyota", 1496, 77, 1290, 68);
 6:
 7:      std::cout << a.model() << "  "
 8:                << a.no() << "  "
 9:                << a.pwratio() << "  "
10:                << a.tax() << std::endl;
11:
12:      std::cout << b.model() << "  "
13:                << b.no() << "  "
14:                << b.pwratio() << "  "
15:                << b.tax() << std::endl;
16:
17:      std::cout << h.model() << "  "
18:                << h.no() << "  "
19:                << h.pwratio() << "  "
20:                << h.tax() << std::endl;
21:
22:      return 0;
23:  }
```

これを実行すると,

```
Skyline  3  5.51471  58000
Civic  5  5.53488  39500
Prius  5  8.89655  0
```

などと表示される.

課題 4.1 [List 4.4]˜ [List 4.7] を入力し, コンパイル・実行せよ.

※ [List 4.1]˜ [List 4.3] はホームページからダウンロード可能.

4.2.3 基底クラスのメンバー関数の呼出し

「ハイブリッドカーは税金がタダ」ではなく,「ハイブリッドカーの税金は, 通常の車の税金の半額」としたいとする. hybrid_car のメンバー関数 tax() を書き換えればいいのだが,「通常の車の税金」の計算をもう一度ここで定義するとコードの重複ができて保守性が悪くなるので,

```
int tax() const {return 「car の tax()」/2;}
```

のように基底クラスの関数を呼び出す書き方がしたい. この,「car の tax()」は, C++ では「car::tax()」と書くことができる.

課題 4.2 上の指示に従って hybrid_car の tax() を書き換え, 実行結果を確認せよ.

4.2.4 継承におけるコンストラクタ/デストラクタ呼び出しの仕組み

- 基本的に, 派生クラスのコンストラクタが起動される際には, まず, その基底クラスのコンストラクタが起動され, その後に派生クラスのコンストラクタが実行される.

- [List 4.5] の 1 行目 hybrid_car のデフォルトコンストラクタが起動される場合, まず基底クラスの car のデフォルトコンストラクタが起動され, その後に hybrid_car のデフォルトコンストラクタが実行される.
- [List 4.5] の 2 行目のように, 基底クラスのどのコンストラクタを起動するか指定があった場合は, そのコンストラクタが起動される. 指定がない場合には, 基底クラスのデフォルトコンストラクタが起動される.

- 派生クラスのデストラクタが起動される際には, 派生クラスのデストラクタの本体が実行され, その後にその基底クラスのデストラクタが起動される.

課題 4.3 car, hybrid_car のコンストラクタ, デストラクタが呼び出される際にメッセージを出力するようにし, [List 4.7] のようなプログラムを実行した際に, どのような順番でコンストラクタ, デストラクタが起動されるか調べよ. ただし, [List 4.7] には hybrid_car のデフォルトコンストラクタの呼び出しがないので, それが呼び出されるように適当な文を追加して調べよ.

4.3 例 2—既存クラスの機能拡張

C++ の継承は「他の誰かが作ったクラスの機能拡張をする」のに便利である. 例として, これまでに作成した stack にアクセスカウンタ (何回データのプッシュやポップや行われたかを記録するカウンタ) を付加することを考える.

元になる stack は 3 章の演習で作ったものとする (どのバージョンでも良い). stack_ac は, これに加えて次の機能を持つものとする.

- int n_push() … スタックが作られてから, 何回 void push() が呼ばれたかを返す.
- int n_pop() … スタックが作られてから, 何回 void pop() が呼ばれたかを返す.

stack_ac は stack の派生クラスとして作ればよい. 変更点は,

- push の回数を数える int 変数 push_count をメンバーに追加する. (必要があれば, 同様に pop_count も.) これらの変数は, 外部から書き換えられないよう, private とする.
- void push(int) を定義しなおす. 内容は, 「基底クラス stack の void push(int) を呼び出し, push_count を 1 増やす 」というものにする.
- 新たなメンバ関数 int n_push() を定義する. 内容は, push_count の値を返すだけ.
- void pop() の呼び出し回数については, 同じようにカウンタを用いて数えてもよいし, sp と push_count の関係から求めてもよい. (後者の場合, sp の値を使うには基底クラス stack において sp を private から protected に変える必要がある. もっとも, sp を直接参照せず, size() を呼び出せば, そのような問題は生じないが.)
- コンストラクタ, デストラクタも, 新たに追加した変数に対して初期化/後処理が必要ならそれを書く.

課題 4.4 上記のアクセスカウンタ付きスタック stack_ac を作れ. プログラムは, 基底クラスの stack, 動作をテストする main(void) とともに示せ.

※ stack_ac のコンストラクタから stack のデフォルトコンストラクタが連鎖的に呼ばれることがあるため, この場合 stack のデフォルトコンストラクタが定義してないとエラーとなる. 引数付きのコンストラクタを 1 つでも定義すると, デフォルトコンストラクタは自動では作られないようなので, stack のデフォルトコンストラクタを陽に定義する必要がある.

コメント

新たなコンテナクラスが欲しくなった場合, 一から定義するよりも, STL のコンテナに機能を追加する形で派生クラスを定義したほうが安全で楽な場合が多い. 例えば, vector において, あらかじめ割り当てられたサイズを越えて [] 演算でアクセスを行った場合にエラーを出すようなクラスは 7 行程度で作れる (参考図書 1 「プログラミング言語 C++ [第 3 版]」の p. 86; ただし, template 構文の勉強が必要.)

4.4 例 3—図形を描くプログラム

継承を用いるとプログラミングが整理される例として, グラフィクスがある. ここでは, 例題として, 点, 長方形, 菱形を描くプログラムを作成する. ただし, グラフィックパッケージを使うのは難しいし, ライブラリの互換性の問題が生じるので, 今回は文字を使った簡単な表示を行うものを作成する.

4.4.1 仕様と単純な実装

次の 4 つのクラスとメイン関数を作成する.

- キャンバス (canvas) … 図形を書き込み, 表示させるためのオブジェクト
- 点 (point), 長方形 (rectangle), 菱形 (diamond) … 図形オブジェクト
- メイン関数 (main)

[List 4.8]

```
1:  #include <iostream>
2:
3:  [canvas の仕様と実装]
4:  [point の仕様と実装]
5:  [rectangle の仕様と実装]
6:  [diamond の仕様と実装]
7:
8:  [main 関数]
```

canvas クラス

文字の 2 次元配列でキャンバスを構成. (白ドット→' ', 黒ドット→'X' とする.)

- クラスの仕様は次の通り.

[List 4.9]

```
 1:  class canvas {
 2:    public:
 3:      canvas(int sx, int sy);  // サイズ sx × sy のキャンバスを構築
 4:      ~canvas();  // デストラクタ
 5:      void clear();  // キャンバスを真っ白にする
 6:      void set(int x, int y);  // 座標 (x,y) にドットを打つ
 7:      void print(std::ostream &os);  // 現在のキャンバスを出力する
 8:    private:
 9:      int size_x;  // サイズ (横)
10:      int size_y;  // サイズ (縦)
11:      char **p;  // 2 次元配列は構築時に動的に割当てる
12:      canvas(const canvas&);   // コピーコンストラクタ禁止
13:      canvas& operator=(const canvas&);  //代入演算禁止
14:  };
```

- << 演算子を使って 出力できるように, 次を追加.

```
15:  std::ostream& operator<<(std::ostream& os, canvas& c) {
16:    c.print(os);
17:    return os;
18:  }
```

- コンストラクタ canvas(int, int) の実装

  サイズを size_x, size_y に設定し, new を使って配列の配列を割当てる

[List 4.10]

```
1:  canvas::canvas(int sx, int sy):  size_x(sx), size_y(sy) {
2:    p = new char*[size_x];
3:    for (int x=0; x<size_x; x++) p[x] = new char[size_y];
4:    this->clear();  // canvas::clear() を呼び出し真っ白に初期化
5:  }
```

- デストラクタ ~canvas() の実装

  配列を動的に割当てているので, デストラクタが必要.

[List 4.11]

```
1:  canvas::~canvas() {
2:    for (int x=0; x<size_x; x++) delete [] p[x];
3:    delete [] p;
4:  }
```

- canvas::clear() の実装

[List 4.12]

```
1:  void canvas::clear() {
2:    for (int x=0; x<size_x; x++) {
3:      for (int y=0; y<size_y; y++) {
4:        p[x][y] = ' ';
5:      }
6:    }
7:  }
```

- canvas::set(int, int) の実装
  - 指定された点に 'X' を書き込むだけ. ただし, キャンバス外の点が指定された場合は何もしない

[List 4.13]

```
1:  void canvas::set(int x, int y) {
2:    if (0<=x && x<size_x && 0<=y && y<size_y) p[x][y] = 'X';
3:  }
```

- canvas::print(std::ostream&) の実装
  - (0,0) が左下となるようにしてある. また「枠」をつけてみました.

[List 4.14]

```
 1:  void canvas::print(std::ostream& os) {
 2:    os << '+';
 3:    for (int x=0; x<size_x; x++) {os << '-';}
 4:    os << '+' << std::endl;
 5:    for (int y=size_y-1; 0<=y; y--) {
 6:      os << '|';
 7:      for (int x=0; x<size_x; x++) {
 8:        os << p[x][y];
 9:      }
10:      os << '|' << std::endl;
11:    }
12:    os << '+';
13:    for (int x=0; x<size_x; x++) {os << '-';}
14:    os << '+' << std::endl;
15:  }
```

point クラス

1 ドットの「点」を表す.

- クラスの仕様は次の通り

[List 4.15]

```
1:  class point {
2:    private:
3:      int px, py;  // 座標
4:    public:
5:      point(int x, int y);  // 座標 (x,y) の「点」を構築
6:      ~point();
7:      void move(int ix, int iy);  // x 方向に ix, y 方向に iy だけ移動
8:      void draw(canvas &c);   // キャンバス c にこの「点」を描く
9:  };
```

- メンバ 関数の実装はどれも簡単

[List 4.16]

```
1:  point::point(int x, int y) :  px(x), py(y) {}
2:  point::~point() {}
3:  void point::move(int ix, int iy) {px += ix; py += iy;}
4:  void point::draw(canvas &c) {c.set(px,py);}
```

rectangle クラス

塗りつぶした長方形を表す

- クラスの仕様は次の通り

[List 4.17]

```
 1:  class rectangle {
 2:    private:
 3:      int px, py;
 4:      int width, height;  //幅と高さ
 5:    public:
 6:      rectangle(int x, int y, int w, int h);
 7:          // 座標 (x,y), 幅 w, 高さ h の長方形を構築
 8:      ~rectangle();
 9:      void move(int ix, int iy);  // x 方向に ix, y 方向に iy だけ移動
10:      void draw(canvas &c);  // キャンバス c にこの「長方形」を描く
11:  };
```

- メンバ 関数の実装→ 課題 4.5 の一部とする.

diamond クラス

塗りつぶした菱形を表わす

- クラスの仕様は次の通り

[List 4.18]

```
 1:  class diamond {
 2:    private:
 3:      int px, py;
 4:      int radius;  //半径 (のようなもの)
 5:    public:
 6:      diamond(int x, int y, int r);
 7:      ~diamond();
 8:      void move(int ix, int iy);
 9:      void draw(canvas &c);
10:  };
```

- メンバ関数の実装

[List 4.19]

```
 1:  diamond::diamond(int x, int y, int r) :  px(x), py(y), radius(r) {}
 2:  diamond::~diamond() {}
 3:  void diamond::move(int ix, int iy) {px += ix; py += iy;}
 4:  void diamond::draw(canvas &c) {
 5:    if (0<=radius) {
 6:      for (int r=-radius; r<=radius; r++) {
 7:        int h = radius - abs(r);
 8:        for (int s=-h; s<=h; s++) {
 9:          c.set(px+r,py+s);
10:        }
11:      }
12:    }
13:  }
```

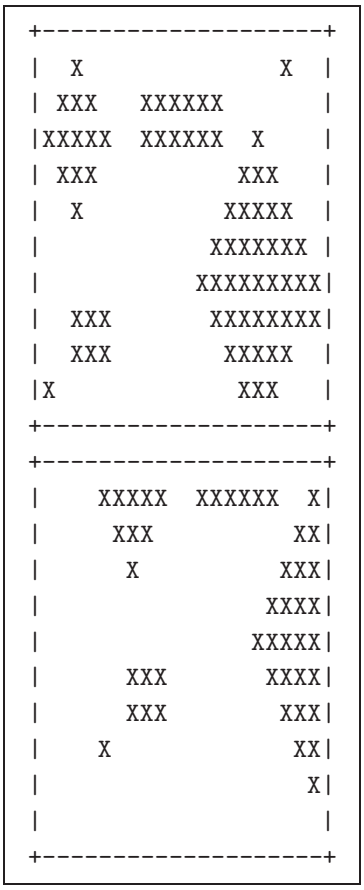
メインプログラム

- 次のようなメインプログラムを考える.
  1. 3 つの点と 2 つの長方形および 2 つの菱形を作る. (3˜ 9 行目)
  2. それをキャンバス c に描き, 出力する. (11˜ 19 行目)
  3. 点, 長方形, 菱形を一斉に (4,2) だけ移動させる. (21˜ 27 行目)
  4. キャンバス c を一旦クリアし, 移動した点, 長方形, 菱形を描き, 出力する. (29˜ 37 行目)

[List 4.20]

```
 1:  int main(void) {
 2:
 3:    point p1(0,0);
 4:    point p2(17,9);
 5:    point p3(19,2);
 6:    rectangle r1(2,1,3,2);
 7:    rectangle r2(7,7,6,2);
 8:    diamond d1(2,7,2);
 9:    diamond d2(15,3,4);
10:
11:    canvas c(20,10);
12:    p1.draw(c);
13:    p2.draw(c);
14:    p3.draw(c);
15:    r1.draw(c);
16:    r2.draw(c);
17:    d1.draw(c);
18:    d2.draw(c);
19:    std::cout << c;
20:
21:    p1.move(4,2);
22:    p2.move(4,2);
23:    p3.move(4,2);
24:    r1.move(4,2);
25:    r2.move(4,2);
26:    d1.move(4,2);
27:    d2.move(4,2);
28:
29:    c.clear();
30:    p1.draw(c);
31:    p2.draw(c);
32:    p3.draw(c);
33:    r1.draw(c);
34:    r2.draw(c);
35:    d1.draw(c);
36:    d2.draw(c);
37:    std::cout << c;
38:
39:    return 0;
40:  }
```

これを実行すると, 次のような図が表示される.



課題 4.5 この [List 4.8] ˜ [List 4.20] のプログラムを完成させよ. `rectangle` クラスの実装は `point` や `diamond` クラスに習って完成させよ. コンパイル・実行し結果を確認せよ.

※ プログラムはホームページよりダウンロードできる.

4.4.2 問題点

- このプログラムでも間違いはないが, 次のような問題がある.
  (1) 全ての描画オブジェクト (点, 長方形, 菱形) を個別の変数に格納し, 個々に `draw(canvas &c)`, `move(int,int)` を呼び出している. 配列等に格納し, 同じメソッドはループで呼び出せるようにしたい.
  (2) 全く同じメンバ関数 `move` が全ての描画オブジェクトに定義されている. もし, アルゴリズム等に変更があった場合には全てに同じ変更を加えなければならない. 何とか定義を一回にできないものか? 同様に, メンバデータ `px`, `py` についても, 同じ宣言が何度も行われている.
- 一つの配列に, 点, 長方形, 菱形をすべて格納できればよいが, 配列には同じクラスの変数しか格納できない. 例えば, `int a[7]` と宣言された整数配列 `a` に対し, `a[0]=3;` と整数を代入することはできても, `a[1] = "Nagisa";` と文字列を代入することはできない. (データのサイズが違うものを配列に混在させるのは不可能.)
- ポインタを用いる方法も考えられるが, C++ では型チェックが厳しいので, `rectangle *a[7]` と宣言した配列に対して, `a[1] = new diamond(2,7,2);` と代入することはできない. cast (型変換) すれば代入が可能な処理系もあるが, 危険だし, 構文的にも面倒なので, 推奨できない.

☆ 継承を用いると, これらの問題が解決できる.

4.4.3 仮想抽象クラス

- 前節の問題は, 新たな「共通」のクラス `obj` を作ることにより解決できる.

1. point, rectangle, diamond の共通点からなるクラス obj を新たに作る.

   例えれば,「猫」「象」「鯨」という 3 つのクラスに対して,「捕乳類」という共通クラスを考えるようなもの.

2. point, rectangle, diamond を obj の派生クラスとして定義する.
3. すると

   (a) 基底クラス「捕乳類」から継承で派生クラス「猫」を作ったとする.「捕乳類」と「猫」はデータのサイズが異なる可能性があるので,「捕乳類」の変数に「猫」のデータを代入することはもちろんできないが, **ポインタなら代入できるようになる**. すなわち,「捕乳類」へのポインタを格納する変数に「猫」のポインタを代入することは許される (継承関係にあれば, 型変換も不要になる). これを利用すると, point, rectangle, diamond のポインタは, obj のポインタ型の変数に代入できるので, 3 つのクラスのポインタを一つの配列に混在して格納することができる.

   (b) 共通部分の定義を一つに集約できるので, move 等の定義の重複を避けることができる.

   というメリットが生じる.

☆ この obj は, 共通部分を集約し, point, rectangle, diamond などの基底クラスになる, という目的のみに使われる. obj クラスのオブジェクトが単独で存在することはないため, このような目的で定義されるクラスのことを**仮想抽象クラス**と呼ぶ.

obj の定義と実装

obj は, 座標 px, py を持つ. また, どのタイプか区別するために, 文字列 type を持つ. ここには, "point", "rectangle", "diamond" のいずれかが代入される. メソッドとしては, コンストラクタ obj の他, move(int,int), draw(canvas&) を持つ.

- obj の定義は次の通り.

[List 4.21]

```
 1:  class obj {
 2:    public:
 3:      obj(int x, int y, const std::string& t);  // コンストラクタ
 4:      ~obj();  // デストラクタ
 5:      void move(int ix, int iy);
 6:      void draw(canvas &c);
 7:    public:
 8:      int px, py;
 9:      std::string type;  // "point", "rectangle", "diamond" のいずれか
10:  };
```

- obj の実装は次の通り.

[List 4.22]

```
1:  obj::obj(int x, int y, const std::string& t) :  px(x), py(y), type(t) {}
2:  obj::~obj() {}
3:  void obj::move(int ix, int iy) {px += ix; py += iy;}
```

draw(canvas&) は個々の派生クラスに対してのみ定義されるべきものなので, obj に対しては定義しない.

point, rectangle, diamond の定義と実装

- point は obj の派生クラスとする. 定義は次のように書き直せる. 共通部分を obj にくくり出すので, 少し簡単になる.

[List 4.23]

```
1:  class point : public obj {  //obj を継承することを宣言している
2:    public:
3:      point(int x, int y);  //コンストラクタ
4:      ~point();  //デストラクタ
5:      void draw(canvas &c);  //描画関数
6:  };
```

- point の実装は次の通り.

[List 4.24]

```
1:  point::point(int x, int y) : obj(x, y, "point") {}
2:  point::~point() {}
3:  void point::draw(canvas &c) {c.set(px,py);}
```

課題 4.6 rectangle, diamond も同様に書き換えよ. メイン関数はそのままで動くはずなので, コンパイル, 実行して結果を確認せよ.

4.4.4 新たな問題

- 継承を用いた定義を用いると, メイン関数 main は次のように簡潔化することができる.

  描画オブジェクトの格納には, vector を用いることにする.

[List 4.25]

```
 1:  int main(void) {
 2:    std::vector<obj*> vec;
 3:
 4:    // 配列に図形要素を放り込む
 5:    vec.push_back(new point(0,0));
 6:    vec.push_back(new point(17,9));
 7:    vec.push_back(new point(19,2));
 8:    vec.push_back(new rectangle(2,1,3,2));
 9:    vec.push_back(new rectangle(7,7,6,2));
10:    vec.push_back(new diamond(2,7,2));
11:    vec.push_back(new diamond(15,3,4));
12:
13:    canvas c(20,10);
14:    // まとめて描画する
15:    for (int i=0; i<vec.size(); i++) vec[i]->draw(c);
16:    std::cout << c;
17:
18:    // まとめて移動する
19:    for (int i=0; i<vec.size(); i++) vec[i]->move(4,2);
20:
```

```
21:    c.clear();
22:    // まとめて描画する
23:    for (int i=0; i<vec.size(); i++) vec[i]->draw(c);
24:    std::cout << c;
25:
26:    // まとめて delete
27:    for (int i=0; i<vec.size(); i++) delete vec[i];
28:    return 0;
29:  }
```

- 残念ながら, **上記の main では正しい結果が得られない!**

  原因は 15 行目および 23 行目の draw の呼び出し

  – 我々としては, vec[i] に格納されているのが point へのポインタであれば point::draw(canvas&)) が呼び出され, rectangle のポインタであれば rectangle::draw(canvas&) が呼び出されることを期待している.

  – しかし, コンパイラの立場に立てば, vec[i]->draw(c) で呼び出すのは obj::draw(canvas&) 以外考えられない! どの関数を呼び出すかはコンパイル時に決めなければならない. どこにサブルーチンジャンプするかコンパイル時に決めないと, コードが生成できないからである. 従って, どのクラスの draw を呼び出すかは, その変数のコンパイル時の型 (静的な型) で決定される. (これに対し, 実際に vec[i] に入っているのが point 型のポインタか rectangle 型のポインタかと言うのは「動的な型」と呼ばれるが, これは実行時に代入が行われて初めて決まり, コンパイル時には決定できない.) この場合だと, vec[i] の型は「obj へのポインタ」だから, vec[i]->draw(c) で呼び出すのは obj::draw(canvas&) と一意に決まる.

課題 4.7 [List 4.20] を [List 4.25] のように書き換え, コンパイル・実行するとどのようなことが起こるか調べよ.

- 通常のコンパイラの枠組では, 実行時に vec[i] にどの型のポインタが入っているかを判定して対応する draw を呼び出すようにするしかない.

  main の 15 行目と 23 行目を次のように書き換えれば, 求める結果が得られる.

[List 4.26]

```
1:  for (int i=0; i<vec.size(); i++) {
2:    if      (vec[i]->type == "point") ((point*) vec[i])->draw(c);
3:    else if (vec[i]->type == "rectangle") ((rectangle*) vec[i])->draw(c);
4:    else if (vec[i]->type == "diamond") ((diamond*) vec[i])->draw(c);
5:  }
```

  vec[i] が一般的な obj 型のポインタではなく, point 型のポインタであることを明示すれば, コンパイラは, vec[i]->idraw に対して point::draw を呼び出すコードを生成してくれる. vec[i] が point 型のポインタであることは, 型変換の構文 (pont*) vec[i] で明示することができる.

  ※ ちなみに, この例のように string の比較で呼び出す関数を切替えるのはあまり推奨できる実装ではないが, 面倒なのでここではサボっている.

課題 4.8 main の 15 行目と 23 行目を上のように書き換え, 実行してみよ.

4.5 仮想関数

4.5.1 draw(canvas&) の仮想関数化

もし呼び出される関数が「動的な型」によって決定される, すなわち, vec[i] にどの型のポインタが代入されているかによって vec[i]->draw(c) で呼び出される関数が決められるなら, [List 4.25] のままで期待通りの結果を得ることができる.

C++ はそのような関数呼び出しの機構を備えている. それが「仮想関数」である.

obj::draw(canvas&) が「仮想関数」として宣言されていたとする. すると,

[List 4.27]

```
 1:  point p(1,3);
 2:  rectangle r(2,1,3,2);
 3:  diamond d(7,7,6,2);
 4:
 5:  obj* x;
 6:  canvas c(20,10);
 7:
 8:  x = &p; x->draw(c);
 9:  x = &r; x->draw(c);
10:  x = &d; x->draw(c);
```

において, x は obj 型へのポインタであるが, x->draw(c) による呼び出しは実行時の型判定によって行われ,

8 行目 … point::draw(canvas&)

9 行目 … rectanble::draw(canvas&)

10 行目 … diamond::draw(canvas&)

が呼び出される.

- obj::draw を仮想関数として宣言するには, obj の draw(canvas&) にキーワード virtual を付けるだけでよい (9 行目) .

[List 4.28]

```
 1:  class obj {
 2:    public:
 3:      obj(int x, int y, const std::string& t);
 4:      ~obj();
 5:      void move(int ix, int iy);
 6:      virtual void draw(canvas &c) = 0;
 7:    public:
 8:      int px, py;
 9:      std::string type;
10:  };
```

- obj 自身に対して draw(canvas&) が呼び出されることがない場合には, 末尾に = 0 を付ける. これは「draw(canvas&) は, obj の派生クラスにおいてのみ定義される」ことを宣言するものである. (このような関数を「純粋仮想関数」と呼ぶ.)

4.5.2 デストラクタの仮想関数化

最後に極めて重要なことを一点. このように抽象仮想クラスを用いる場合には, **デストラクタを必ず仮想関数にしておかなければならない**ことに注意.

[List 4.29]

```
 1:  class obj {
 2:    public:
 3:      obj(int x, int y, const std::string& t);
 4:      virtual ~obj();
 5:      void move(int ix, int iy);
 6:      virtual void draw(canvas &c) = 0;
 7:    public:
 8:      int px, py;
 9:      std::string type;
10:  };
```

[List 4.25] の最後に `delete vec[i]` を行うと, ポインタ `vec[i]` によって指されるオブジェクトのデストラクタが呼ばれる. `vec[i]` が `point` を指す場合には `point` のデストラクタが起動されるべきだが, `~obj()` を通常の関数にしておくと静的な型判断により, `~obj()` が起動されてしまう.

`~obj()` を仮想関数にすれば, 動的な型判断により呼出しが行われるので, `~point()` が起動される.

☆ ついでだが, `~obj()` を仮想関数にして `~point()` が起動された場合も, デストラクタの連鎖 (chaining) によって, `~point()` の実行後 `~obj()` が実行されることになる.

課題 4.9 `obj::draw(canvas&)` と `obj::~obj()` を仮想関数にし, [List 4.25] のままの `main` 関数でどのような結果が得られるか調べよ.

コメント

- ここでの例のように, ポインタの動的な型 (実行時に決まる型) に依存して呼び出す関数を変える仕組みを “dynamic dispatch” と言う. また, 同じ (抽象基底クラスの) オブジェクトが動的な型により様々な振舞をすることをポリモーフィズムと言う.
- dynamic dispatch の実現のメカニズムはコンパイラにより異なるが, 一般的な実装法でのオーバヘッドはそれほど大きくない.
- 「継承」が C++ の本領であることに間違いはない. これにより, 大規模なプログラムが整理された形で開発できると言われている.
- 継承の利点と弊害については様々な議論があるが, dynamic dispatch やポリモーフィズムに関しては継承の有効な利用法であると言われている.
- ぱっと見には「便利な C」位にしか見えない C++ だが, その背後にはプログラミングの種々の概念が存在していることが分かる. がむしゃらにプログラムを書くことも必要だが, こうしたプログラミングのセンスも必要であることを意識して欲しい.

Nagisa ISHIURA