C から入る C++

担当: 金丸隆志

第 11 回

1 はじめに

前回の授業において, 論理回路シミュレータのソースを提示し, それを動かしてみた.

今回と次回の演習において, そのプログラムを理解するための知識を補足する.

内容は「**コンストラクタ初期化子**」,「**継承**」,「**仮想関数**」,「**連想配列**」,「**二分木**」などを予定している.

2 基本的な動作の確認

配布したプログラムは論理回路の簡易シミュレータである. まず本章ではこのプログラムの利用法を簡単に述べる.

2.1 使用法

シミュレータを利用するには, simMain.cpp ファイルを編集する必要がある. 配布した simMain.cpp には図 1 のように回路が定義されている.

```
Edge i1, i2, i3;
Edge o3;
Edge o1, o2;

And and1( o1, i1, i2 );
Or or1( o2, o1, i3 );
Not not1( o3, o2 );
```

図 1: 回路の定義.

このリストによって, 以下のことがわかる.

- Edge, And, Or, Not というクラスがソースファイルのどこかで定義されている.
- And, Or, Not クラスのインスタンス and1, or1, not1 を作成するとき, コンストラクタに Edge クラスのインスタンスを引数として与えている.

実は, 上記の定義によって以下のような論理回路を定義したことになる.

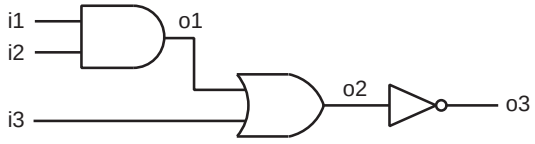

図 2: 定義された論理回路.

次に, simMain.cpp のうち, 以下の部分に着目しよう. これは simulation クラスのインスタンス sim

```
sim.set(0, 0, i1);
sim.set(0, 0, i2);
sim.set(0, 0, i3);

sim.set(10, 1, i3);
sim.set(25, 0, i3);

sim.set(30, 1, i1);
sim.set(35, 1, i2);
```

図 3: 入力信号の定義

のメンバ関数 set を呼び出し, 入力信号を定義している. 定義された入力信号は図 4 の通り.

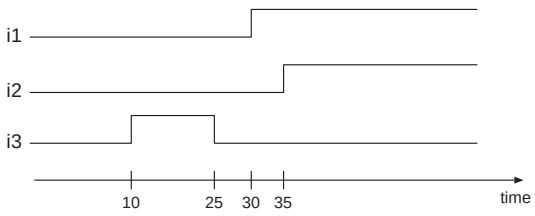

図 4: 定義された入力信号.

このプログラムを実行すると, o1, o2, o3 の出力がコンソールに表示される. なお, 論理素子の「出力計算」, および「次の素子への結果の伝搬」にはそれぞれ 1 ステップの時間がかかる仕様になっていることに注意しておく.

2.2 common.h, common.cpp の概説

さて, 前節で扱ったような論理回路シミュレータを作成するためには, 論理素子や結線に対応するクラスが必要である. それを定義しているファイルが「common.h」および「common.cpp」である.

これらのファイルではいくつかのクラスが定義されているが, 主要なクラスは「Edge」, 「Pin」, 「Node」であり, それらを模式的に表したのが図 5 である.

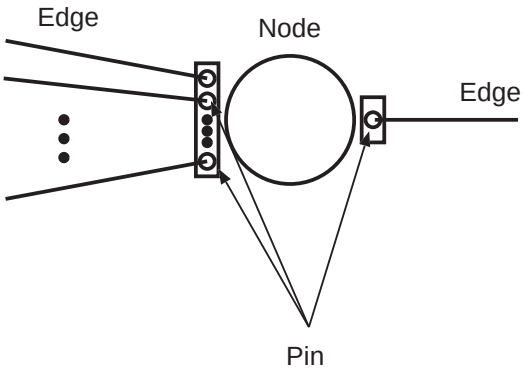

図 5: Edge, Pin, Node の模式図.

「Node」は論理回路の構成要素, すなわち And, Or, Not などを意味するが, 抽象的なクラスとして実装されており, その機能を**継承**することで And, Or, Not クラスを実現していることに注意しよう.

2.3 今回の資料の流れ

以上, 配布した論理回路シミュレータを利用するための最低限の知識について触れた. 以後, common.h および common.cpp を理解するための知識を補う方向で演習をすすめる.

ただし, 各自が考察する余地を残すため, ソースを一行ずつ解説するようなことはしないので, 以下の解説がソースのどこで使われているか考えてみて欲しい.

なお, 以下でとりあげるのは論理回路シミュレータ以外でも広く使われる一般的な内容であるため, 是非身に付けて欲しい.

[**課題 1**]
simMain.cpp を編集して半加算器を実現し, その動作を確認せよ.
伝搬の遅延が蓄積するので, 入力はゆっくりと変化させると良い.
なお, 半加算器がどのような動作をする回路か分からない場合は, 論理回路の授業の内容を復習すること.

3 コンストラクタ初期化子

myComplex クラスを思いだそう. myComplex クラスのコンストラクタは図 6 のようになっており, データメンバ x (real part) および y (imaginary part), にそれぞれ xx, yy を代入することで初期化を行っていた.

myComplex.h 内部

```
...
public:
  myComplex(double xx=0, double yy=0){
      x = xx; y = yy;
  } /* コンストラクタ */
  ...
```

図 6: myComplex クラスのコンストラクタ.

しかし, この方法では const (定数) のデータメンバの初期化はできない. それに対し, 図 7 のように**コンストラクタ初期化子** (constructor initializer) を用いれば const のデータメンバを初期化できる. (ただし, myComplex クラスのデータメンバは const ではないので, どちらの方法でも初期化できる [1])

```
  ...
public:
  myComplex(double xx=0, double yy=0) :
  x(xx), y(yy){ } /* 実際は一行. もちろん, 適当な位置で改行してもよい */
  ...
```

図 7: コンストラクタ初期化子によるデータメンバの初期化.

また, **継承** (inheritance) を用いる場合, **派生クラ**

ス (derived class) 内で **基底クラス** (base class) の初期化を行いたい場合にも上記のコンストラクタ初期化子が用いられる.

4 継承 (inheritance)

一般に, オブジェクト指向プログラム言語 (OOPL) は以下の機能を実装していなければいけないと言われる.

- カプセル化 (データ隠蔽の一種)
- 継承 (inheritance)
- 多態 (polymorphism)

本章ではこのなかの**継承**に焦点をあてる. なお, 多態を理解するためにも継承の概念が必要であるため, OOPL を身に付けるには, 継承の理解が必須であるといえるだろう.

4.1 継承とは何か (1)〜汎化と特化

まず, クラスを始めて学んだ時に取り上げた「テレビ」クラスを取り上げよう. テレビクラスのデータメンバには「電源の状態」,「現在のチャンネル」, 現在の音量」などが考えられ, メンバ関数には「電源 ON/OFF」,「チャンネル変更」,「音量調節」等があったことを思いだそう.

ここで, 同様な例として,「BS チューナ内蔵テレビ」クラス,「インターネット接続機能つきテレビ」クラスを考えよう. これらが持つべきデータメンバやメンバ関数は例えば図 8 のようになるだろう. こ

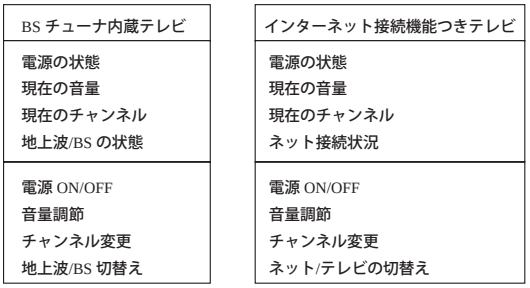

図 8: 「BS チューナ内蔵テレビ」クラスと「インターネット接続機能つきテレビ」クラス.

れを見ればわかるように, 二つのクラスには共通のメンバが多く, これらをそれぞれ別途にプログラミングするのは無駄が多い.

そこで, 図 9 のように「BS チューナ内蔵テレビ」クラスと「インターネット接続機能つきテレビ」クラスが,「テレビ」クラスの機能をして受け継いで実現できると便利である. このように, あるクラスの機能を受け継いで別のクラスをつくる機能を**継承** (inheritance) と呼ぶ. ここで, 元になる「テレビ」

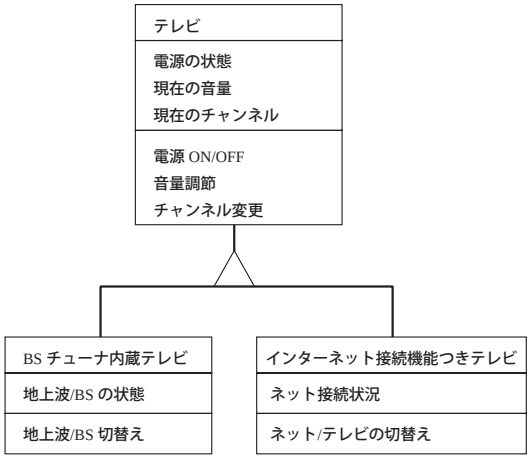

図 9:「テレビ」クラスの機能を継承した「BS チューナ内蔵テレビ」クラスと「インターネット接続機能つきテレビ」クラス.

クラスは**基底クラス** (base class) と呼ばれ,「テレビ」クラスを受け継いだ 2 つのクラスは **派生クラス** (derived class) と呼ばれる.

継承を用いたプログラミングには, 例えば以下のようなメリットがある.

- 追加する機能のみをプログラミングしてゆくため, プログラムの記述量が減る (**差分プログラミング** (incremental programing) [3]).
- 基底となるクラスに手を加えることがないため, バグが発生したときに問題の切り分けがしやすい.
- 基底クラスに変更を加えた時 (たとえば「コントラスト調節」機能を追加するなど), 全ての派生クラスにその変更が反映される.

なお, ここで扱った継承の例は「汎化 (generalization)」と「特化 (specialization)」という言葉で理解することができる [2]. まず, 図 8 の 2 つのクラスから共通の性質を括り出して基底クラスとなる「テ

レビ」クラスを導き出した. これは 2 つのクラスの汎化であると言える. 逆に, 基底クラスである「テレビ」クラスから出発して, 特殊な機能を追加して「BS チューナ内蔵テレビ」クラスと「インターネット接続機能つきテレビ」クラスを導くこともでき, これはクラスの特化であると言える.

4.2 継承とは何か (2)〜カスタマイズ

前節では「汎化と特化」としての継承の例を見たが, 本節では「クラスのカスタマイズ」としての継承の例を紹介する [2].

以前紹介した, 「遺伝的アルゴリズム (Genetic Algorithm: GA)」クラスを思いだそう. GA クラスは図 10 のようなメンバを持つと考えられる.

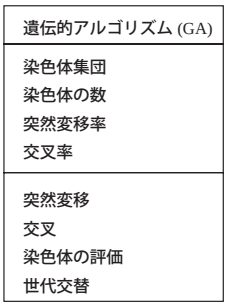

図 10: 「遺伝的アルゴリズム (GA)」クラス.

実は, GA をある問題に適用するとき「染色体の評価」というメンバ関数が重要な役割を果たす. 例えば,「染色体の評価」関数を変更するだけで「画像認識用 GA」,「音声認識用 GA」など, 様々な問題に GA を適用することができる [4]. これは「GA」クラスをカスタマイズすることで「画像認識用 GA」クラスや「音声認識用 GA」を作り出せることを意味するが, この様な場合にも継承の機能を用いることができる.

継承を用いた場合の「GA」クラス,「画像認識用 GA」クラス,「音声認識用 GA」の関係を図示すると図 11 のようになる [5]. 注意すべきことは,「染色体の評価」という関数が基底クラスと派生クラスの両方に存在することである. これは派生クラスで「染色体の評価」という関数を**再定義 (オーバーライド)** し, カスタマイズを実現していることに相当する.

以上は OOPL の一般論であるが, C++ で関数の再定義を行う場合, 再定義される関数 (すなわち「染色体の評価」) は基底クラス内で**仮想関数** (virtual function) として定義しなければならない [6]. これを行わないと,「画像認識用 GA」クラスから基底クラス側の「染色体の評価」が呼び出されることもあり, 思わぬバグにつながることがあるため, 注意しなければならない.

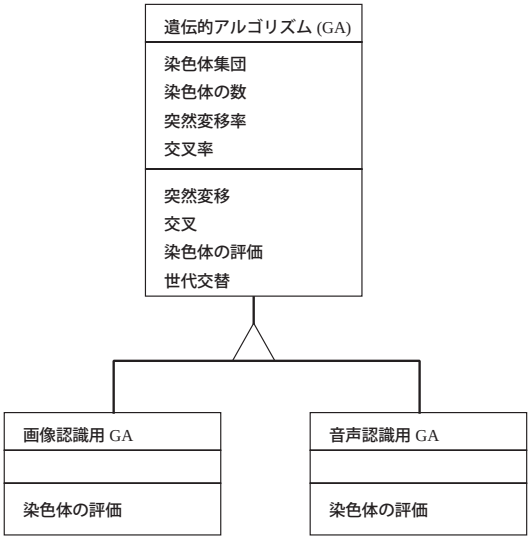

図 11: 「GA」クラス,「画像認識用 GA」クラス,「音声認識用 GA」の関係.

4.3 継承の実現方法

以上, 継承に関する一般論を行った. 本節では C++ で継承を実現するための方法を簡単に解説する.

図 12 のような関係にある Base クラスと Derived クラスを考えよう. Derived クラスは, データメンバ z と メンバ関数 func2 が追加され, メンバ関数 func1 を再定義していることに注意しよう.

Base クラスと Derived クラスを記述すると, 図 13 のリストのようになるであろう. このリストのポイントは以下の通り.

- 再定義によって上書きされる関数 func1 には, Base クラス内で virtual というキーワードをつけて**仮想関数**とする.
- 仮想関数の存在するクラスのデストラクタにも virtual をつける [7].
- 「class Derived」の後に「: public Base」と書くことで, Base クラスの派生クラスであるこ

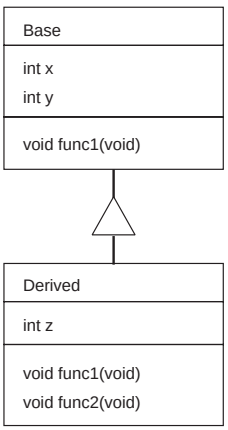

図 12: Base クラス と Derived クラス.

とを宣言する [8].

- Derived クラスのコンストラクタで Base クラスのデータメンバを初期化するには, コンストラクタ初期化子を用いて Base クラスのコンストラクタを呼び出す.

[**課題 2**]

(1) common.h, common.cpp を見て, 図 13 の様な構造を見つけよ. そして, 図 9, 11, 12 の様な図を描いてみよ (ソースの全てを理解する必要はない). その際, 2.2 節の記述を参考にすること.

(2) 身近なもので継承のメカニズムを用いることができる例を考えよ. プログラミング可能かどうかは考慮しなくて良い.

参考文献

[1] 文献 [2] のように, 図 6 のタイプの初期化ではなく, 図 7 のようなコンストラクタ初期化子を用いた初期化を推奨する文献もあることに注意しておく.

[2] Tucker!, “憂鬱なプログラマのためのオブジェクト指向開発講座,” 翔泳社 (1998).

[3] 柴田望洋, “C プログラマのための C++ 入門,” ソフトバンク (1992).

[4] L. デービス 編, “遺伝アルゴリズムハンドブック,” 森北出版株式会社, (1994).

```
class Base{
   private:
      int x;
      int y;
   public:
      /* コンストラクタ */
      Base(int xx=0, int yy=0) : x(xx), y(yy){}
      virtual ~Base(){} /* デストラクタ */
      virtual void func1(void);
};

class Derived : public Base{
   private:
      int z;
   public:
      /* コンストラクタ */
      Derived(int xx=0, int yy=0, int zz=0) :
Base(xx,yy), z(zz){} /* 実際は一行 */
      void func1(void); /* 再定義 */
      void func2(void);
};

void Base::func1(void){
   ...
}
void Derived::func1(void){
   ...
}
void Derived::func2(void){
   ...
}
```

図 13: C++ による継承の記法

[5] もちろん, 画像認識用 GA には認識対象となる画像がデータメンバに必要だし, 音声認識用 GA には音声信号が必要であるが, 図では省略した.

[6] GA クラスを用いる場合, 必ず派生クラスを作ってから具体的な問題に適用させる. すると, 基底クラス内の「染色体の評価」関数は必ず再定義で上書きされるため, 実体は必要はない. そのような場合, 基底クラスでの「染色体の評価」関数は**純粋仮想関数**として実現するのが自然である. 図 13 の例で言うと, 純粋仮想関数

は「virtual void func1(void)=0;」などと宣言するだけで良い.

[7] 仮想関数の存在するクラスのデストラクタに virtual をつける理由は CMagazine 2002 年 6 月号の「C++ 基礎講座」等を参照.

[8] public 以外のアクセス指定子もつけることができるが, その効果は C++ の教本を参考にして欲しい.