C++プログラミングII

第1回 クラスの基本

岡本秀輔

成蹊大学理工学部

はじめに

プログラミング的思考とは

- 将来どのような進路を選択しどのような職業に就くとしても、普遍的に求められる力
- 自分が意図する一連の活動を実現するために、どのような動きの組合せが必要であり、一つ一つの動きに対応した記号を、どのように組み合わせたらいいのか、記号の組合せをどのように改善していけば、より意図した活動に近づくのか、といったことを論理的に考えていく力
  - 小学校プログラミング教育の手引（第三版）
  - 令和２年２月文部科学省
  - https://www.mext.go.jp/content/20200218-mxt_jogai02-100003171_002.pdf

文章詳細な読解

- 自分が意図する一連の活動を実現するために、
  どのような動きの組合せが必要であり、
  - 必要な作業の把握:
  - 何と何をすれば目的を達成できる?
  - 細かいことは抜きにしてどんな作業が必要？
- 一つ一つの動きに対応した記号を、
  どのように組み合わせたらいいのか、
  - 目的達成のための作業に対して：
  - 組み合わせ可能な手続きの把握
  - 現状でできることと組み合わせの方法
- 記号の組合せをどのように改善していけば、
  より意図した活動に近づくのか、
  - 組み合わせ方の改善
  - 組み合わせ方はいろいろ

講義実験のねらい

- C++プログラミング I で学んだこと
  - 基本データ型と構造体（型の組み合わせ）
  - 代入、条件分岐、繰り返し（基本的な動作）
  - 関数（基本動作をまとめて名前をつける）
    - 関数呼び出しの組み合わせ
    - 小さいプログラムはこれで良い
- 大規模なプログラムで必要のこと $\rightarrow$ 階層化
  - 階層間のやりとりの仕方
  - 階層ごとの責任の明確化
- 授業の目標
  - クラスの基本
  - STL コンテナ, イテレータ, アルゴリズム
  - データ構造の基本的な使用方法（一部作り方）

オブジェクト指向プログラミング

構造体のプログラム例1

- 構造体を使うと複数のデータを一つにまとめられる
- 構造体と関数だけでは作業の階層化が見えにくい

```
struct Robot{ int x, y; };
Robot a, b;
move(a);  // a に関すること
move(b);  // b に関すること
if (bump(a, b))... // a,b 両者に関すること?
```

オブジェクト指向プログラミング

- 関連するデータとそれらを操作する関数をまとめる
  - 関連データ間で一貫性を保持（無矛盾の運用）
  - 責任の所在を明確にする（特定データ専用の処理）
  - 他への影響を減らす（交換可能な実装）
- 専門用語:
  - インタフェースと実装, カプセル化, 情報隠蔽
  - 抽象データ型、継承、多様性など

Robotのオブジェクト化（イメージ）

- 構造体に専用の関数を導入
- 一般の関数とメンバ関数の区別により階層化

```
struct Robot {
    int x, y;
    void move(); // x,yを変更する専用の関数
};
Robot a, b;
a.move();  // aに関すること
b.move();  // bに関すること
if (bump(a, b))... // a,b両者に関すること?
```

C++クラスの概要

- クラスはユーザ定義型
  - 自作の型の変数が宣言できる
  - 変数はオブジェクトやインスタンス変数とも呼ばれる
- クラスはメンバの集まり
  - データメンバとメンバ関数
  - 他の言語ではメンバ関数をメソッドと呼ぶ
- メンバ関数には準備用や後片付け用の関数もある
- メンバはドット (.) やアロー (->) 演算子でアクセス
- クラス用の演算子 (+, ! など) を定義できる
- クラスはメンバの名前空間である
  - メンバの名前がクラスの外側で制限とはならない
- `public` メンバがクラスのインタフェース, `private` メンバが実装の役割
- 構造体はメンバがデフォルトで `public` なクラス

The C++ Programming Language, 4th ed より

public（公開）と private（非公開）の必要性

銀行のシミュレーションを考える

- インタフェース:（口座利用者に公開）
  - 口座を開く方法、閉じる方法
  - お金の預け方、下ろし方
  - 送金の指定方法
- 実現に必要なこと, 実装: (口座利用者には非公開)
  - 金庫の取扱い
  - 金庫内の金額管理
  - 他行との送金に関するやりとり

- 銀行の内部を口座利用者に見せない（情報隠蔽）ことで、口座利用と銀行内の運用を分離可能。
- 口座利用のプログラムはインタフェースだけを使って銀行を利用できる（カプセル化）。
- 設計変更によって銀行内部処理も取り替えも可能。

C++のクラス

c++17 の使用

- この講義実験では C++17 の使用を前提としている。
- g++で a.cpp をコンパイルと実行をするには以下のようにする。

```
$ g++ -std=c++17 a.cpp
$ ./a.out
```

- 性能を計測する際には最適化を施してコンパイルする。
- 以下の-O は最適化 (Optimize) の指定である。

```
$ g++ -O -std=c++17 a.cpp
$ ./a.out
```

構造体のプログラム例2

```
#include <iostream>
struct TimeData0 { int min{}, sec{}; };

// 加算後に sec<60 を保証
void add(TimeData0& t, int m, int s) {
    t.min += m;
    t.sec += s;
    if (t.sec >= 60) {
        t.min += t.sec/60;
        t.sec %= 60;
    }
}

int main() {
    TimeData0 t;
    add(t, 3, 50);
    std::cout << t.min <<":"<< t.sec <<"\n";
}
```

C++クラスで書き換える

```
class TimeData1 {
  public:
    int min{}, sec{}; // データメンバ
    void add(int m, int s) { // メンバ関数
       min += m;
       sec += s;
       if (sec > 60) {
          min += sec/60;
          sec %= 60;
       }
    }
};

int main() {
   TimeData1 t; // min/sec を内部に持つ
   t.add(3, 50);  // t の min/sec を変更
   std::cout << t.min <<":"<< t.sec <<"\n";
}
```

メンバ関数

- min と sec はクラスのデータメンバ
- 実体は変数 t の内部にある
- t.add(3, 50) がメンバ関数 add の呼び出し
- メンバ関数 add では
  - 仮引数 m と s
  - min と sec は呼び出し時の変数に対応
  - t.add(3, 50) ならば t.min と t.sec を変更

```
    void add(int m, int s) {
        min += m;
        sec += s;
        ...
    }


...
    TimeData1 t; // min/sec を内部に持つ
    t.add(3, 50);   // t の min/sec を変更
...
```

考察

- add() をメンバ関数にした
  - 加算時に sec < 60 を保証できる
  - 負の数の対応も add() の修正できそう
  - TimeData1 の内部処理を明確にできるかも
- 考慮事項:
  - min, sec は main で自由に変更してよいか？
  - min，sec へのアクセスに制限は必要か?
  - 時間の表示形式を一貫して同じに保てるか?

```
t.sec = 65; // これを禁止する必要があるか？
cout << t.min <<":"<< t.sec <<"\n"; // 問題ない？
```

アクセス指定子

- メンバの公開非公開を指定する
- private ラベルより下側のメンバは非公開
- public ラベルより下側のメンバは公開
  - class 定義ではデフォルトでメンバが非公開

```
class TimeData2 {
  private: // 省略可能
     int min{}, sec{};  // 非公開
  public:
     void add(int m, int s) { // 公開
       ...
     }
     void print() { ... } // 公開（出力形式の責任を担う）
};

...
TimeData2 x;
x.add(3,10); // OK
x.print();   // OK
x.min = 1;   // error:  非公開メンバへのアクセス
```

構造体とクラスの関係

- struct と class では無指定メンバの公開属性が異なる
  - struct では無指定が公開メンバ
  - class では無指定が非公開メンバ
- それ以外はほぼ同じ
- どちらも public, private ラベルの指定が何度でも可

```
struct TimeData_s {
    int min{}, sec{};
};

class TimeData_c {
    int min{}, sec{};
};
...
TimeData_s s;
s.min = 10; // OK
TimeData_c c;
c.min = 10; // error
```

ゲッター／セッター

- アクセス指定子でメンバを非公開にした
  - `min`, `sec` を個別に取り出す方法がほしい
  - 設定もしたいが `sec` の設定時には処理が必要
- ゲッター／セッターメンバ関数
  - 設定と取り出しに特化したメンバ関数
  - アクセスメソッドとも呼ばれる
  - クラス構築のテクニック
    - 作らなくても良い
    - 特殊な文法があるわけでもない
    - `get` や `set` で始まる名前をつける習慣がある

```
x = t.get_s();  // getter
t.set_s( x + 40 ); // setter
```

getter/setterの導入

```
class TimeData3 {
    int min{}, sec{};
    void normalize() { // 内部で使う非公開メンバ関数
        if (sec >= 60) {
            min += sec/60; sec %= 60;
        }
    }
  public:
    void add(int m, int s) {
        min += m; sec += s; normalize();
    }
    void set_s(int s) {
        sec = s; normalize();
    }
    void set_m(int m) { min = m; }
    int get_s() { return sec; }
    int get_m() { return min; }
    void print() { ... }
};
```

その他の特徴

const の問題

- 構造体の引数にはリファレンス指定 (&) が多い
- 引数の内容を変更しない場合には const も指定する
- クラスの変数も同様の方針でよい
- コンパイラは以下の例でエラーを出力する

```
void print_remain(const TimeData3& t)
{
    std::cout <<"残り時間は:";
    t.print();
}
```

エラー: passing ‘const TimeData3’ as ‘this’ argument discards qualifiers [-fpermissive]

- 理由：t.print() の呼び出しが t を変更するかもしれない
  - コンパイラには判断がつかない
  - ’this’ が t のポインタを示すが。。。（不親切）

const メンバ関数

- データメンバを無変更の関数には const を指定
  - getter や print 関数などが対象

```
class TimeData4 {
    int min{}, sec{};
    ...
  public:
    ...
    int get_s() const { return sec; }
    int get_m() const { return min; }
    void print() const { ... }
};
...
void print_remain(const TimeData3& t)
{
   std::cout <<"残り時間:";
   t.print();  // OK: print() は const メンバ関数
}
```

メンバ関数の宣言

- メンバ関数は宣言と定義を分けられる
  - 通常の関数と同じ考え方
  - 大規模プログラムではファイルの分割で利用する
    - 複数人で開発する場合に有効
- 以下はメンバ関数の宣言のみのクラス
  - インタフェースを明示した形になる
  - ヘッダファイルとして別ファイルに分離できる
    - 例えば、`timedata.hpp` というファイル名にする

ソースコード 1: timedata.hpp

```
#include <string>

class TimeData5 {
    int min{};
    int sec{};
  public:
    void add(int m, int s);
    std::string str() const;
};
```

メンバ関数の定義

- 所属するクラス名を関数名の前に指定する

ソースコード 2: timedata-impl.cpp

```
#include <sstream>
#include <iomanip>
#include "timedata.hpp" // 自作には二重引用符
void TimeData5::add(int m, int s) {
    min += m;
    sec += s;
    if (sec >= 60) {
        min += sec/60;
        sec %= 60;
    }
}
std::string TimeData5::str() const {
    std::ostringstream o;
    o << min <<":"<<
    std::setw(2)<< std::setfill('0')<< sec;
    return o.str();
}
```

クラスの使用

- クラスの実装の詳細が別ファイルに分離されている
- string や vector はこの使い方をしている

ソースコード 3: main.cpp

```
#include <iostream>
#include "timedata.hpp"
void print(const TimeData5& t)
{
    std::cout <<"経過時間:"<< t.str() <<"\n";
}

int main()
{
    TimeData5 t;
    t.add(3, 50);
    print(t);
    t.add(1, 15);
    print(t);
}
```

分割したファイルのコンパイル

- まとめて指定する場合
  - 必要ファイルを指定する
  - timedata.hpp は#include で取り込まれる
  - -o オプション（小文字のオー）で実行ファイル名の指定

```
$ g++ -std=c++17 main.cpp timedata-impl.cpp
$ ./a.out
$ g++ -std=c++17 main.cpp timedata-impl.cpp -o prog1
$ ./prog1
```

- 別々にコンパイルしてまとめる場合
  - -c オプションの指定: .o の拡張子のファイル作成
  - オブジェクトファイルと呼ぶ（機械語プログラム）

```
$ g++ -std=c++17 -c main.cpp
$ g++ -std=c++17 -c timedata-impl.cpp
$ g++ main.o timedata-imple.o -o prog1a
$ ./prog1a
```

まとめ

まとめ

- オブジェクト指向プログラミング
  - データと手続きをまとめて責任を明確にする
  - ソフトウエアの実装を階層化できる
  - C++のクラスはこれをサポートするもの
- C++のクラス
  - データメンバとメンバ関数
    - メンバ関数はデータメンバを操作する
  - `public/private` でアクセス制御
    - getter/setter が必要になることもある
    - const メンバ関数が必要になることも
  - メンバ関数の宣言と定義の分離
    - クラスのインターフェースが明確になる
    - 分割コンパイルで有効

用語

以下の用語を確認せよ

- プログラミング的思考
- インタフェースと実装
- クラスとオブジェクト
- メンバ関数, const メンバ関数
- public ラベル, private ラベル
- ゲッターとセッター, アクセスメソッド
- ヘッダファイル
- 分割コンパイル, オブジェクトファイル