C++プログラミング II

第 2 回 コンストラクタ/演算子多重定義

岡本秀輔

成蹊大学理工学部

コンストラクタ (constructor)

コンストラクタ（構築子）とは

- オブジェクトが作られる際に実行される関数
  - 変数宣言のみで実行される（明示的な呼び出しなし）
  - オブジェクトの初期化処理を担う
  - 例：変数宣言時に何らかの処理を行う
- 特徴
  - クラス名と同じ名前の関数
  - 引数を変えて多重定義を作っても良い
- メンバ関数との違い
  - 戻り値の指定がない (return 文なし)
  - 関数本体を const 指定しない
  - メンバ初期化リストの指定（後述）

```
class Loud {
  public:
    Loud() { std::cout << "Hello!!\n"; }
};
int main() {
   Loud a;
}
```

確認

1. main() 関数を空にして、クラス Loud の大域変数を宣言してみよ。
2. 以下の vector 配列を宣言してみよ。
   - `std::vector<Loud> x(10); // 要素 10 個`

コンストラクタの定義

- 戻り値の型を書かない
- 引数は通常の関数と同じ
- データメンバを操作する

```
class TimeData6 {
public:
    int min{}, sec{};
    TimeData6(int m, int s) {
        min = m + s/60;
        sec = s%60;
    }
};

int main() {
    TimeData6 t{3, 50};
    std::cout << t.min <<":"<< t.sec <<"\n";
}
```

コンストラクタの多重定義

- 一般関数、メンバ関数と同様に多重定義ができる
- メンバ関数を呼び出しても良い（共通に使える）
- 他のコンストラクタを呼び出せない
  - コンパイルはできるが、オブジェクトの対象が変わる

```
class TimeData7 {
    int min, sec;
public:
    TimeData7(int s) { set(0, s); }
    TimeData7(int m, int s) { set(m, s); }
    void set(int m, int s) {
        min = m + s/60;
        sec = s%60;
    }
};
int main()
{
    TimeData7 t1{3, 50};
    t1.set(4, 70);
}
```

コンストラクタ定義の副作用

- 引数付きコンストラクタを定義すると
  そのままでは初期値なしの変数宣言ができなくなる

```
class TimeData8 {
public:
    int min{1}, sec{30}; // 初期値を指定しているが
    TimeData8(int m, int s) {
        min = m + s/60;
        sec = s%60;
    }
};
int main()
{
    TimeData8 t1{3, 65}; // 4:05
    std::cout << t1.min <<":"<< t1.sec <<"\n";
    // TimeData8 t2; // エラーとなる
}
```

デフォルト コンストラクタ

- 引数なしのコンストラクタの名称
  - オブジェクトのデフォルト値を決める
  - 変数宣言などで必要になることが多い
- 自動生成（特別扱い）:
  - デフォルト コンストラクタは自動生成の対象
  - 自作コンストラクタがあると自動生成されない
- 自動生成されないことへの対処（以下のどれか）

```
// 明示的に作る
TimeData8() { min = sec = 0; }
// またはデフォルト引数で代用する
TimeData8(int m =0, int s =0) { min = m; sec = s; }
// またはコンパイラに生成を依頼する
TimeData8() =default;
```

コンストラクタと右辺値

- コンストラクタはオブジェクトの右辺値を作る
- コンストラクタが他のコンストラクタを呼べない理由
  - 別のオブジェクトができるだけのため

```
void func(MyClass i)
{
    std::cout << i.get() <<"\n";
}

int main()
{
    MyClass x; // デフォルト コンストラクタが呼ばれる
    x = MyClass(123); // int 引数のコンストラクタ
    x = MyClass{123}; // 中括弧でも同じ
    std::cout << x.get() <<"\n";
    func( MyClass(345) ); // 実引数にも指定可能
}
```

コンストラクタの宣言と定義の分離

- クラス定義の中にコンストラクタの宣言
- クラス定義の外でコンストラクタの定義
- 他のメンバ関数と同じくクラス名と::を指定
- 戻り値の型はないことに注意する

```
class T {
  public:
    T(int); // 宣言
    ...
};

T::T(int x) { ... } // 定義
```

メンバ初期化子リスト

- コンストラクタの動作前にメンバを初期化する方法
  - データメンバが const やリファレンスの場合には必須
  - データメンバのコンストラクタ呼び出しにも利用
- 指定方法と注意点：
  - 仮引数と関数本体の間に、コロンではじめてカンマ区切りの初期化子（変数と初期値の対）を並べる
  - データメンバの宣言した順に並べる必要がある
  - int などの通常の変数も指定できる（省略しても良い）
  - 初期化の順番のため他のメンバ関数を呼ばない方が良い

```
class T {
    int abc, xyz;
  public:
    T(int x) :abc{x},xyz{0} {
      // 関数の本体
    }
};
```

初期化子リストの例

- max は const なので関数本体より前に初期化が必要
- val は基本データ型だが初期化子に入れても良い
- ival は初期値指定があるのでリストからもれても良い
- data はコンストラクタで s 個の要素 (値 0) となる

```
class Foo {
    const int max;
    double val;
    int ival{3};  // 初期化子リストからもれても良い
    std::vector<int> data;
public:
    Foo(int x,double v,int s) :max{x},val{v},data(s) {
        // 必要ならば他の処理
    }
};
```

今は詳細を知らなくても良い関数

ファイルやメモリをクラスで管理する際に利用される。
興味のある人は調べるとよい。

- デストラクタ (破壊子)
  - オブジェクトが削除される時に呼ばれる関数
  - 変数のスコープが終わった、一時変数の破壊時
  - cout を指定してデバッグに使うことがある
- コピー コンストラクタ
  - 同一クラスのオブジェクトをコピーする
  - 引数の値渡し、戻り値、変数の複製、一時変数など
- 代入演算子の多重定義
  - 代入文で呼ばれる関数
  - コピー コンストラクタと同じ処理が必要となる

一般の仕事との関係

- コンストラクタとデストラクタ
  - 仕事は始めと終わりが大切
- コピーコンストラクタ、代入演算子
  - 仕事は引き継ぎや委託の時に慎重になるべき

演算子の多重定義

モチベーション

- 構造体と同様にクラスの変数も大小を比較したい

```
TimeData x, y;
if (x < y) ....  // x.less(y) より直感的
```

- 同様にクラスの変数が cin, cout に対応すると良い

```
TimeData x;
cin >> x;
cout << x;
```

  - 出力は str() メンバ関数があれば十分？
  - `cout << x.str();`

- 数値が関係するクラスでは演算が指定できると良い

```
TimeData x, y, z;
z = x + y; // z = x.add(y) より分かりやすい
z += 30;   // 30 秒追加; z.add(30) でも良い？
```

単項演算と二項演算

- 演算の種類
  - `+a`，`-b` などを単項演算と呼ぶ
  - `a+b`，`a*b`，`a<b` などを二項演算と呼ぶ
- オペランドを変更する演算
  - 数学ではほとんどの単項演算や二項演算でオペランドを変更しない
  - C++ではオペランドを変更する演算子がある
    - `++`，`--`，`+=`，`-=`，`...`

演算子の多重定義 (オーバローディング)

- オブジェクトを引数にした特殊な名前の関数を作る
  - `operator<` :二項演算
  - `operator>>` :二項演算
  - `operator+` :単項演算と二項演算
  - `operator+=` :二項演算
- クラス用の演算子多重定義の仕方は二種類ある
  - 一般の関数として多重定義
  - メンバ関数として多重定義
  - どちらかを選択する

```
// 一般関数での多重定義の例
bool operator<(const TimeData& x, const TimeData& y)
{
  return x.getseconds() < y.getseconds();
}
```

メンバ関数での演算子多重定義

- 二項演算用には右側のオペランドのみ引数
- 単項演算子には引数なしの関数を作る
- 左オペランドは自分自身

```
class TimeData9 {
    int min{}, sec{};
  public:
    TimeData9(int m, int s): min(m), sec(s){}
    int getseconds() const { return min*60 + sec; }

    bool operator<(const TimeData9& y) const {
      return getseconds() < y.getseconds();
    }
};

int main() {
  TimeData9 a{2, 19}, b{3, 59};
  if (a < b) std::cout << "a is small\n";
}
```

一般関数での多重定義

- 二項演算用には二つの引数の関数を作る
- 単項演算子には一つの引数の関数を作る
- <<や>>では左オペランドが cin や cout ならばメンバ関数での演算子多重定義はできない
- 一般関数は private メンバにアクセスできない
  - public メンバ関数を作る
  - 特別にアクセスを許可する (フレンド関数)

```
// t.str() を通して文字列を取り出す
ostream& operator<<(ostream& out, const Data& t)
{
   return out << t.str(); // クラス Data は str() を持つ
}
```

フレンド

- private データメンバへのアクセス許可与える方法
- クラスの中で許可する相手を friend に指定する
  - 許可された関数は private にアクセス可
- friend 指定の対象
  - フレンド関数：一般関数・他クラスのメンバ関数
  - フレンドクラス：指定クラスの全メンバ関数

```
class TimeData10p {
    int min{}, sec{};
public:
    TimeData10p(int m, int s):min{m},sec{s}{}
    // friend キーワードで print 関数に許可を与える
    friend void print(const TimeData10p& t);
};

void print(const TimeData10p& t) {
    // 引数 t の private メンバにアクセス
    std::cout << t.min <<":"<< t.sec <<"\n";
}
```

フレンド関数の取り込み定義

- クラス内でのフレンド関数の定義
- 一般関数もヘッダファイルに指定したい時に便利
  - インライン関数と呼ばれる形式になる

```
class TimeData10p {
    int min{}, sec{};
public:
    TimeData10p(int m, int s):min{m},sec{s}{}
    // print は一般の関数で定義も書いてある
    friend void print(const TimeData10p& t) {
        std::cout << t.min <<":"<< t.sec <<"\n";
    }
};
```

注意：

- フレンド関数は特別に許可する手段
- なるべく使わない方が良い

フレンド関数で多重定義

- <<による出力は一般関数となる
- str() のようなメンバ関数がない場合

```
class TimeData10 {
    int min{}, sec{};
public:
    TimeData10(int m, int s):min{m},sec{s}{}
    friend std::ostream&
    operator<<(std::ostream& out,const TimeData10& t){
        return out << t.min <<":"<< t.sec;
    }
};

int main() {
    TimeData10 t{1, 30};
    std::cout << t <<"\n";
}
```

例:TimeData

- 以下のクラスと main() から演算子多重定義を考える

```
class TimeData11 {
    int min{}, sec{};
  public:
    TimeData11(int m, int s)
        :min{m+s/60}, sec{s%60}{}
    int getm() const { return min; }
    int gets() const { return sec; }
    std::string str() const {
      std::ostringstream o;
      o << min <<":"<< std::setw(2)
        << std::setfill('0')<< sec;
      return o.str();
    }
};
int main() {
   TimeData11 x{3, 35}, y{2, 10};
   std::cout << x+y <<"\n"; // 5:45
   std::cout << x-y <<"\n"; // 1:25
}
```

TimeData の演算子多重定義

- 可能な限り friend にしない方が良い
- 省略の仕方もいくつかある

```
auto   // 戻り値の型は auto で省略できる
operator+(const TimeData11& a, const TimeData11& b)
  { return TimeData11{a.getm()+b.getm(),
                      a.gets()+b.gets()}; }

TimeData11 // return で型名を省略した場合
operator-(const TimeData11& a, const TimeData11& b)
  { return {a.getm()-b.getm(),a.gets()-b.gets()}; }

auto&   // auto でも & は省略しない
operator<<(std::ostream& out, const TimeData11& t)
  { return out << t.str(); }
```

演算子多重定義の選択指針

- 一般関数にする（できれば friend 指定もしない）
  - 四則演算でメンバを変更しない場合
    - 単項演算も含めて考える
    - friend やゲッターを嫌うならばメンバ関数でも良い
  - 入出力で左オペランドが対象クラスではない場合
  - ライブラリなどメンバ関数にできない場合
- メンバ関数とする
  - データメンバを更新する (+=, ++などの) 場合
  - 演算子が =，[]，()，-> の場合

まとめ

コンストラクタ

- オブジェクトが作られる際に実行される関数
- クラス名と同じ名前の関数
- メンバ関数の一種であるがわずかに違いある
  - メンバ初期化リスト
  - `return` なし、関数本体に `const` 指定なし
- デフォルト コンストラクタ
  - クラスのデフォルト値を決める役割
  - 自動生成の対象だが、行われない条件がある
- コンストラクタの呼び出し指定
  - オブジェクトの右辺値となる

演算子の多重定義

- `operator<`のような名前の関数
- 演算子によって引数の数が決まっている
  - 単項演算、二項演算
- 多重定義の仕方は二種類ある
  - 一般関数、メンバ関数
- フレンドの指定
  - 多重定義に関わらずメンバ関数以外の関数に `private` のアクセス権を与える

演習1の予告

以下のページのプログラムを完成させる

- 演習 1: p.3, p.5, p.19, p.24-25
- 演習 2: p.7 のクラスを修正しコンパイラにデフォルトコンストラクタを生成依頼する。t2 を宣言して、中身を出力する。