

[prog2] Programming C++ (C6) Exercise Guide (Ex13)

11/20, Monday 3rd period.

Ex13 について

▶ 多態性 (polymorphism)

Polymorphismの種類

▶ 関数定義のオーバーロード (※Ex06で既にやった)

⇒ 同じ関数の名前で、異なるプロトタイプ関数を同時に実装できる

▶ 関数定義のオーバーライド

⇒ 全く同じ型、引数、名前の関数だが、内部処理を再定義できる

▶ 演算子定義のオーバーロード

⇒ クラスごとに演算子の意味・操作を再定義できる

関数定義のオーバーロード（例題 1）

• overload.cc

▶ プロトタイプが異なる関数

void func(void);

int func(int);

string func(string,string);

void func(double,double);

は同時に存在できる。

プログラム内に書かれた関数
呼び出しに応じて、どの関数の
ことかをコンパイラが判断する

▶ ただし、**int func (void);**

のように、**引数が同じものは
コンパイラが判断できずエラーに。**

⇒ このような実装はできない

```
1  #include <iostream>
2  #include <string>
3
4  using namespace std;
5
6  //5つの異なるバージョンがあるfunc()
7  void func(void);//ver. A
8  //int func(void);//ver. B これは ver. Aと引数が被ってコンパイラが判別できずエラーに
9  int func(int);//ver. C
10 string func(string,string);//ver. D
11 void func(double,double);//ver. E
12
13
14 int main(){
15     int a, x;
16     string s;
17
18     func();// ver. Aが呼ばれる
19     a = func(3); // ver.Cが呼ばれる
20     s = func("I never","fail");//ver. Dが呼ばれる
21     func(30.3,3.1);// ver. Eが呼ばれる
22
23     // ver. B は ver. Aと共存不可能。
24     // for(int i =0; i<100; i++) x = func();
25     //cout << x << endl;
26
27     cout << a << endl;
28     cout << s << endl;
29
30     return 0;
31 }
32
33
34
35 void func(void){// ver. A
36     cout << "Sorrow World." << endl;
37 }
38
39 /*
40 int func(void){// ver. B
41     static int i;
42     return ++i;
43 }
44 */
45 int func(int x){// ver. C
46     return x*x;
47 }
48
49 string func(string a,string b){ // ver. D
50     string s = a + " " + b;
51     return s;
52 }
53
54 void func(double x,double y){ // ver. E
55     cout << "x+y=" << x+y << endl;
56 }
```

関数定義のオーバーライド（例題2）

- ▶ 関数を再定義 ⇒ 特にクラスの継承で活躍

- ▶ 例題2はAを基底クラスとして

B、DがAのサブクラス

CがBのサブクラスになるよう継承

⇒ それぞれに同名、かつ内部処理の異なる関数を実装
という前提で考える。

基底A

|

+-B

|

| +-C

|

+-D

関数定義のオーバーライドの1（例題2）

- 各クラスに、内部処理が異なる **double calc()** を実装

```
1  #include <iostream>
2
3  using namespace std;
4
5  class A{
6  protected:
7      double a,b;
8
9  public:
10     A() : a(12.5), b(8.0){}
11     double calc(void){
12         cout << "a+b" << endl;
13         return a+b;
14     }
15 };
```

```
43  int main(){
44      A a;
45      B b;
46      C c;
47      D d;
48
49      cout << a.calc() << endl;
50      cout << b.calc() << endl;
51      cout << c.calc() << endl;
52      cout << d.calc() << endl;
53
54      return 0;
55 }
```

- main関数から、各クラスオブジェクトの**calc()**を呼ぶと？

```
17  class B : public A{
18
19  public:
20     double calc(void){
21         cout << "a-b" << endl;
22         return a-b;
23     }
24 };
25
```

```
26  class C : public B{
27  public:
28     double calc(void){
29         cout << "a*b" << endl;
30         return a*b;
31     }
32 };
```

```
33  class D : public A{
34  public:
35     double calc(void){
36         cout << "a/b" << endl;
37         return a/b;
38     }
39 };
40
```

関数定義のオーバーライドの1（例題2）

- 各クラスで再定義されたcalc()が確かに実行されている

```
1  #include <iostream>
2
3  using namespace std;
4
5  class A{
6  protected:
7      double a,b;
8
9  public:
10     A() : a(12.5), b(8.0){
11         double calc(void){
12             cout << "a+b" << endl;
13             return a+b;
14         }
15     };
17     class B : public A{
18     public:
19         double calc(void){
20             cout << "a-b" << endl;
21             return a-b;
22         }
23     };
24
26     class C : public B{
27     public:
28         double calc(void){
29             cout << "a*b" << endl;
30             return a*b;
31         }
32     };
33
34     class D : public A{
35     public:
36         double calc(void){
37             cout << "a/b" << endl;
38             return a/b;
39         }
40     };
```

- とりあえず、問題無し

- override1.ccの実行結果

```
./a.out
a+b
20.5
a-b
4.5
a*b
100
a/b
1.5625
```

関数定義のオーバーライドの2（例題2）

- ▶ `double calc()` を各々実装するが、直接呼出ししない
- ▶ クラスAに`call()`という関数を実装し、`call()`を通して間接的に`calc()`を呼び出す設計になったとすると...？
⇒ `call()` は継承されるので、全てのサブクラスにもある

```
5  class A{
6  protected:
7      double a,b;
8
9      double calc(void){
10         cout << "a+b" << endl;
11         return a+b;
12     }
13
14 public:
15     A() : a(12.5), b(8.0){}
16
17     double call(void){
18         return calc();
19     }
20
21 };
```

call()はcalc()を呼ぶためだけに作られたインターフェース

呼び出しは`call()`で!!

```
49  int main(){
50      A a;
51      B b;
52      C c;
53      D d;
54
55      cout << a.call() << endl;
56      cout << b.call() << endl;
57      cout << c.call() << endl;
58      cout << d.call() << endl;
59
60      return 0;
61 }
```

関数定義のオーバーライドの2（例題2）

▶ 全部結果が同じになった...

- `override2.cc`の実行結果

```
$/a.out  
a+b  
20.5  
a+b  
20.5  
a+b  
20.5  
a+b  
20.5
```

▶ `call()` はクラスAで実装されている

⇒ 「クラスAに実装された`calc()`を呼び出す」`call()`関数としてサブクラスへ継承されている

⇒ 関数定義のオーバーライドが効いていない！

関数定義のオーバーライドの2（例題2）

- ▶ じゃあ、`call()`も各クラスで再定義すれば
 - ⇒ もちろん、それは動く
 - ⇒ 本質的でない部分のコードを汚すことは、オブジェクト指向やクラス継承といった抽象的な考え方から逸脱する
 - ⇒ 無意味にコードの可読性も下げる
- ▶ `call()`関数を変えないで、想定通りの呼び出し（サブクラスでオーバーライドした関数を実行）はできるか？

関数定義のオーバーライドの3（例題2）

▶ **virtual** キーワードの利用

- ▶ 関数定義の前に **virtual** と書くことで、その関数は「**仮想関数**」となり、クラス継承によって再定義されることを想定した関数になる

- ▶ こうすることによって **call()** で間接的に呼び出された **calc()** 関数は各クラスで再定義された **calc()** 関数となる！

```
5  class A{
6  protected:
7      double a,b;
8
9      virtual double calc(void){ //仮想関数化
10         cout << "a+b" << endl;
11         return a+b;
12     }
13
14 public:
15     A() : a(12.5), b(8.0){}
16
17     double call(void){
18         return calc();
19     }
20
21     };
```

関数定義のオーバーライドの3（例題2）

▶ ちゃんと動いた(1例目と同じ結果になった)

- `override3.cc`の実行結果

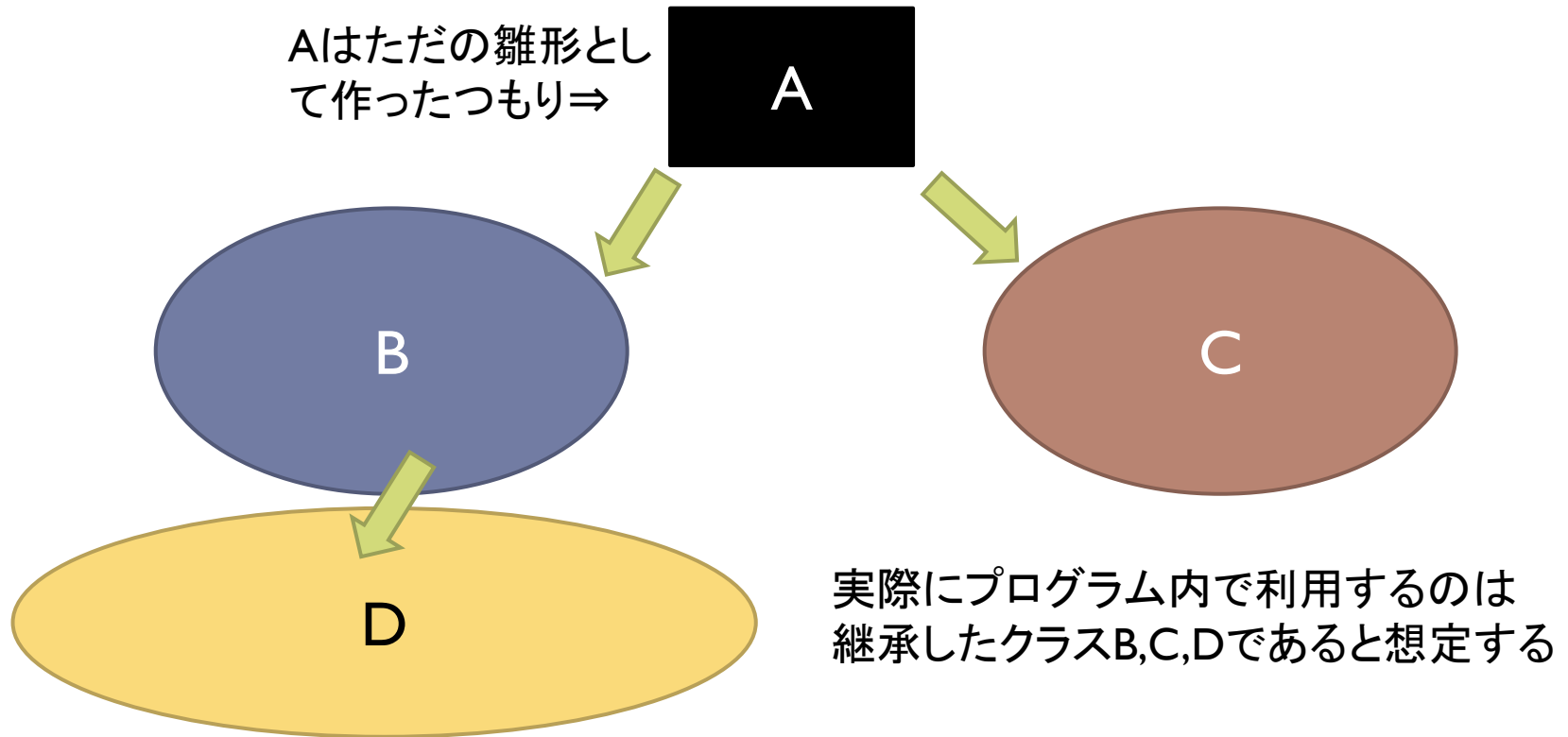
```
$/a.out  
a+b  
20.5  
a-b  
4.5  
a*b  
100  
a/b  
1.5625
```

▶ `virtual` キーワードは、「継承先で再定義があった場合には、そちらを呼び出せ」という指示を出すことだと考えられる

⇒ インターフェースを経由して、内部のメソッド(関数)を呼び出すような実装を、継承していくような場合、**virtual** は必要不可欠に

関数定義のオーバーライドの4（例題2）

- ▶ virtual キーワードを用いた**抽象クラス**の実装
- ▶ 「クラスAは直接実体を作成して利用するわけではなく、継承して使われることを前提に作られたクラス」だとする



関数定義のオーバーライドの4（例題2）

- ▶ 継承先のクラスで同じ名前の関数を持たせる
- ▶ クラスAで先に定義して継承させたい
⇒ が、クラスAは直接使用するつもりはないので、
中身の実装はしなくていいんだけど...
- ▶ 継承先で(再)定義されることを前提として、関数のプロトタイプ宣言だけを書く ことができる

```
5 | class A{ //純粋仮想関数を含むので抽象クラスとなり、実体を生成できない
6 |     protected:
7 |         double a,b;
8 |
9 |         //純粋仮想関数（継承先のクラスで定義される）
10 |         virtual double calc(void)=0;
11 |
12 |     public:
13 |         A() : a(12.5), b(8.0){}
14 |
15 |         double call(void){
16 |             return calc();
17 |         }
18 |
19 |     };
```

関数定義のオーバーライドの4（例題2）

▶ **virtual** double calc(void) = 0;

“calc()はクラスAでは使うつもりがないので、型だけ定義”

⇒ 「継承先でちゃんと定義する」ことを前提

▶ **virtual関数のプロトタイプ = 0;**

の形で宣言されるひな型を**純粋仮想関数**という

```
5 | class A{ //純粋仮想関数を含むので抽象クラスとなり、実体を生成できない
6 |     protected:
7 |         double a,b;
8 |
9 |         //純粋仮想関数（継承先のクラスで定義される）
10 |         virtual double calc(void)=0;
11 |
12 |     public:
13 |         A() : a(12.5), b(8.0){}
14 |
15 |         double call(void){
16 |             return calc();
17 |         }
18 |
19 | };
```

関数定義のオーバーライドの4（例題2）

純粋仮想関数を使用する時の注意

- ▶ 「あくまで継承して使われる」ことを大前提とした「抽象クラス」という扱いをされる

⇒ 純粋仮想関数を含むクラス定義は実体化できない

```
47 int main(){
48     //A a; // 純粋仮想関数を含むため、「抽象クラス」となりインスタンス化ができない（コメントアウトを外すとエラーが出る）
49     B b;
50     C c;
51     D d;
52
53     //cout << a.call() << endl;
54     cout << b.call() << endl; //継承したクラスでは、正しく動く
55     cout << c.call() << endl;
56     cout << d.call() << endl;
```

「ちゃんと継承しろよ〜」（約束）

関数定義のオーバーライドの4（例題2）

純粋仮想関数を使用する時の注意

- ▶ 「あくまで継承して使われる」ことを大前提とした「抽象クラス」という扱いをされる

⇒ 純粋仮想関数を含むクラス定義は実体化できない

```
47 int main(){  
48     //A a;    // 純粋仮想関数を含むため、「抽象クラス」となりインスタンス化ができない（コメントアウトを外すとエラーが出る）  
49     B b;  
50     C c;  
51     D d;  
52  
53     //cout << a.call() << endl;  
54     cout << b.call() << endl; //継承したクラスでは、正しく動く  
55     cout << c.call() << endl;  
56     cout << d.call() << endl;
```

あるクラスを「ひな型」として継承して、似たような構造のサブクラスを量産するような場合に使っていくことになる

関数定義のオーバーライドの4（例題2）

実行結果は想定通り。

⇒ クラスAは「抽象クラス」となり、クラス変数として実体化はできない

⇒ 継承したクラスB,C,Dでは各々が定義した関数が実行されている

- `override4.cc`の実行結果

```
./a.out  
a-b  
4.5  
a*b  
100  
a/b  
1.5625
```

あるクラスを「ひな型」として継承して、似たような構造のサブクラスを量産するような場合に使っていくことになる

演算子定義のオーバーロード（例題3）

定義したクラス内の演算子の役割を書き換える

以下の演算は常識的な範疇

$$3 + 5 = 8$$

$$\text{"ab"} + \text{"cd"} = \text{"abcd"}$$

自分で演算子のルールを定義すれば、次のようにも変更可

$$3 + 5 = 1$$

$$2 + (-6) = 3$$

$$\text{"abc"} + \text{"def"} = \text{"adbefc"}$$

様々な構造を持つクラス

⇒ クラス構造に合わせた演算子があった方が便利

⇒ template 等と組み合わせると、より抽象的・柔軟な定義が必要

演算子定義のオーバーロード（例題3）

整数の演算のルールを変える

⇒ カレンダー上の曜日に整数値を対応(0,1,2,...,5,6)させる特殊な状況下を考える

ex) 1週間=7日より

3日 と 17日 は同じ曜日

12日 と 19日 は同じ曜日

⇒ $3 == 17$, $12 == 19$ となる世界

特殊な演算を持つ構造のために、
整数値を1つだけ持つクラス
Integerを作る。

```
5  class Integer{
6      private:
7          int x;
8      public:
9          bool operator==(Integer y){
10             if(x%7 == y.x%7){ return true; }
11             else{return false; }
12         }
13         bool operator!=(Integer y){
14             if(x%7 != y.x%7){ return true; }
15             else{return false; }
16         }
17         int operator+(Integer y){
18             return (x + y.x)%7;
19         }
20         int operator-(Integer y){
21             return (x - y.x)%7;
22         }
23         void setx(int v){ x = v; }
24     };
```

演算子定義のオーバーロード（例題 3）

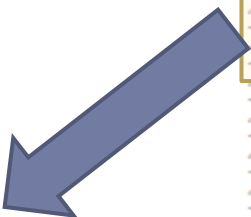
当然、普通の 比較演算子 `==` では
`3 != 17`, `12 != 19` であるが...

private:

int x;

```
bool operator==(Integer y){  
    if(x%7 == y.x%7){ return true; }  
    else{ return false; }  
}
```

```
5  class Integer{  
6      private:  
7          int x;  
8      public:  
9          bool operator==(Integer y){  
10             if(x%7 == y.x%7){ return true; }  
11             else{return false; }  
12         }  
13         bool operator!=(Integer y){  
14             if(x%7 != y.x%7){ return true; }  
15             else{return false; }  
16         }  
17         int operator+(Integer y){  
18             return (x + y.x)%7;  
19         }  
20         int operator-(Integer y){  
21             return (x - y.x)%7;  
22         }  
23         void setx(int v){ x = v; }  
24     };
```



自分でルールを書くと、`==` の意味を変えられる

（ 7 で割った余りが同じときに「等しい」）

⇒ `a.x=3`, `b.x=17` のとき, `if(a==b)` は真になる

演算子定義のオーバーロード（例題3）

（二項）演算子定義の構文

⇒「演算子」の名前の関数を定義すると考えればよい

戻り値の型 operator【定義する演算子】(引数){...}

このスタイルは実質「クラスのメンバー関数の再定義」の形式

Integer a, b; としたとき、if(a==b) とすると、

```
bool operator==(Integer b){ //これは a.operator==(b)みたいな呼出
    if(x%7 == b.x%7){ return true; }
    else{ return false; }
}
```

If文の左辺値のクラスオブジェクトのメンバー関数として == 演算子を呼び出して、右辺値を引数として受け取る

左辺値 (Integer a) のメンバー関数と呼んでいるため、

operator内のif文 “x”%7とは、a.x%7 のことである

演算子定義のオーバーロード（例題3）

クラスのメンバーとしてルールを定義できる演算子の代表例

・二項演算

代入

=

A+B の + のような四則演算と剰余

+, -, *, /, %

不等号

<, ==, >, !=

論理

&&, ||

配列の添字

[]

・単項演算（引数を void とするメンバー関数として定義）

アドレスの値参照

*

アドレス参照

&

インクリメント、デクリメント

++

...などなど、というか殆どが再定義可能

演算子定義のオーバーロード（例題3）

クラスのメンバーとしてルールを定義できない例

- throw 例外スロー
- sizeof() 変数のサイズをとる演算子
- . クラスのメンバーへのアクセス (a.hoge() 等の .)

自分で新しい演算子は作れない

ex) @ という記号を決めて、 a@b など

演算子定義のオーバーロード（例題 3）

加法 + を 減法 - にも変えられる

$a + b$ を $a - b$ として処理する例。

コンストラクタで $a=7$, $b=5$ として

$a + b = 2$ となることを確認。

```
#include <iostream>

using namespace std;

class Int{
private:
    int x;
public:
    Int(int in): x(in){}
    int operator+(Int b){
        x = x - b.x;
        return x;
    }
};

int main(){
    Int a(7), b(5);
    int c;
    c = a+b;
    cout << c << endl;
    return 0;
}
```


演算子定義のオーバーロード（例題3）

参考)

グローバル関数として、**演算子を定義することもある程度可能**
⇒ クラス内定義よりもややこしい、制約が多い

「クラス設計の延長で」、ということでこの授業では
クラス内の演算子の定義・拡張というテーマに絞る

“演算子” = “クラスのメンバー関数”

⇒ 2項演算の場合は、**引数は演算相手の値を1つ**

⇒ 単項演算の場合は、**void型（クラスメンバーから持ってくる）**