

継承とオーバーライド

Game Programming B #02

向井 智彦

ゲームプログラミングAのおさらい

- クラス, メンバ変数, メンバ関数
- アクセス指定子とカプセル化
- コンストラクタ
- オーバーロード
- 継承
- 仮想関数
- オーバーライド

クラス：複数の変数と関数の複合体

```
1.  class Vector2
2.  {
3.      public:
4.          Vector2(double ix, double iy);
5.      public:
6.          double Length() const;
7.          void Print() const;
8.      public:
9.          double x, y;
10. };
```

メンバへのアクセス

1. `Vector2 v;` // Vector2変数vの宣言

2. `v.x = 10;`

// v + ピリオド + 要素名 x

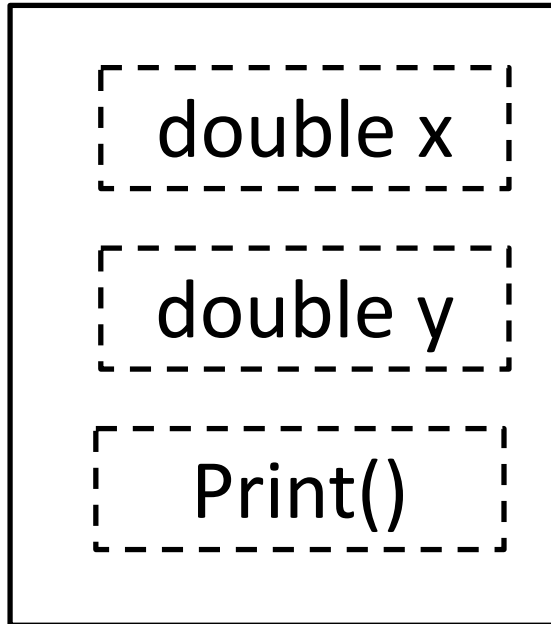
3. `v.y = 20;`

// v + ピリオド + 要素名 y

4. `v.Print();`

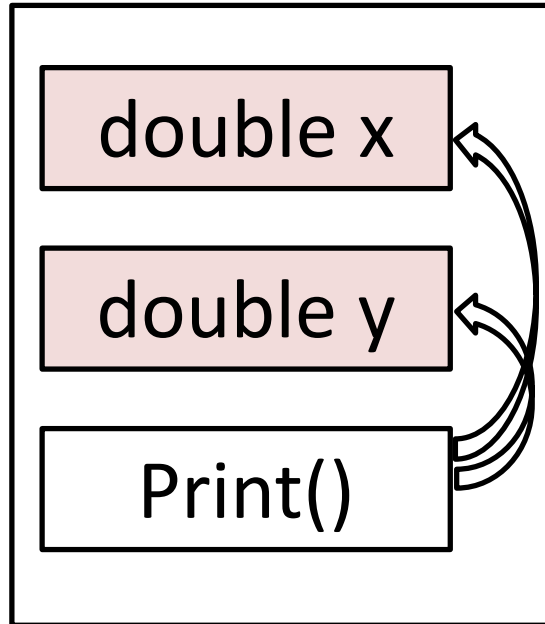
// v + ピリオド + 関数名 Print

クラス - オブジェクト - メンバ



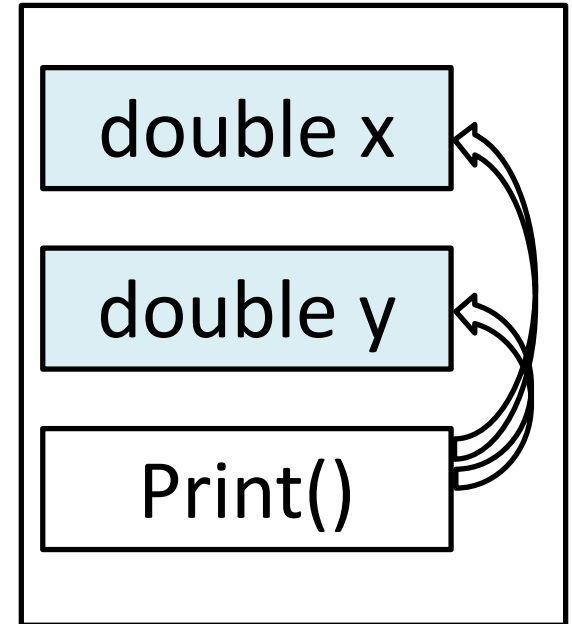
`class Vector2`

クラスはメンバの数や
種類を定義



`Vector2 a;`

クラス変数=オブジェクト・インスタンスを宣言すると
インスタンスごとにメンバ変数が用意される
メンバ関数は、各インスタンスがもつメンバ変数に
アクセスできる

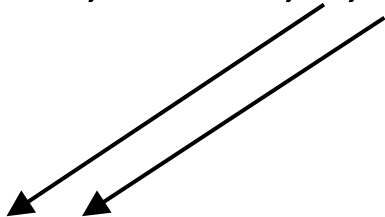


`Vector2 b;`

Vector2::Print()

```
class Vector2
{
public:
    void Print() {
        printf("%f, %f¥n", x, y);
    }
public:
    double x, y;
};
```


各オブジェクトのメンバ変数




```
int main()
{
    Vector2 a, b;
    a.x = 10;  a.y = 5;
    b.x = 0;   b.y = 20;

    a.Print(); // 10, 5
    b.Print(); // 0, 20
    return 0;
}
```

ベクトルaの成分x, yをprint



ベクトルbの成分x, yをprint



Vector2::Length()

```
class Vector2
{
public:
    double Length() {
        return sqrt(x * x + y * y);
    }
private:
    double x, y;
};
```

```
int main()
{
    Vector2 a, b;
    a.x = 10;  a.y = 5;
    b.x = 0;   b.y = 20;

    cout << a.Length(); //11.18
    cout << b.Length(); //20.0
}
```

アクセス指定子

使わせたい
メンバは
publicに

隠したい
メンバは
privateに

この部分も
publicに

```
class PC {  
    public:  
        void TypeKey();  
        void ClickMouse();  
    private:  
        CPU c;  
        Memory mem;  
        SSD s;  
        GPU g;  
    public:  
        ....  
}
```


カプセル化

```
class Vector2
{
private:
    double x, y;
public:
    void Print()
    {
        printf("%f, %f¥n", x, y);
    }
private:
    void PrivateFunc() { .... }
};
```

メンバ関数内ではメンバ変数へのアクセス制限無し

クラス内の他のメンバ関数から呼び出し可能
一方、クラス外からのアクセスは不可

```
int main()
{
    Vector2 a, b;
    a.x = 10;  a.y = 5;
    b.x = 0;   b.y = 20;

    a.Print(); // OK
    b.Print(); // OK
    a.PrivateFunc(); // エラー
}
```

全てエラー:
クラス外から
privateメンバは
アクセスできない

引数付きコンストラクタ

```
class Vector2
{
private:
    double x, y;
public:
    Vector2() {
        x = 0.0; y = 0;
    }
    Vector2(double initX,
            double initY) {
        x = initX; y = initY;
    }
};
```

デフォルト
コンストラクタ

```
int main()
```

```
{
```

変数名だけの時は
デフォルトコンストラクタ

```
    Vector2 a;
```

```
    Vector2 b(); // エラー
```

```
    Vector2 b(2.0, 3.0);
```

```
    Vector2 c = Vector2(0, 1.0);
```

```
}
```

引数付きの場合は
数・種類が一致する
コンストラクタが呼ばれる

関数オーバーロード

```
class Vector2
```

```
{
```

```
    ....
```

```
public:
```

```
void Init() { x = 0; y = 0; }
```

```
void Init(double a)
```

```
{ x = a; y = a; }
```

```
void Init(double a, double b)
```

```
{ x = a; y = b; }
```

```
int main()
```

```
{
```

```
    Vector2 a;
```

```
    a.Init();
```

```
    a.Init(0.0);
```

```
    a.Init(1.0, 0.0);
```

```
}
```

const メンバ関数

```
class Vector2
```

```
{
```

```
private:
```

```
    double x, y;
```

```
public:
```

```
    double GetX() const
```

```
{
```

```
    return x;
```

```
}
```

```
};
```

「このメンバ関数はオブジェクトの
内部状態=メンバ変数を変更しない」
ことを宣言する

クラスの継承

基本クラス

```
class BaseClass1
{
private:
    int x;
public:
    int GetX() {return x;}
    void SetX(int ix){x=ix;}
};
```

派生クラス

```
class DerivedClass1
    : public BaseClass1
{
private:
    int y;
public:
    int GetY() {return y;}
    void SetY(int iy) {y=iy;}
};
```

DerivedClassはメンバ変数xとメンバ関数GetX, SetXを継承
→DerivedClassはGetX, SetX, GetY, SetYを提供

クラスの継承 contd.

```
int main()
{
    BaseClass1 base;
    DerivedClass1 derived;

    base.SetX(10);
    derived.SetX(10); // 基本クラスのGetXを継承
    derived.SetY(20);
    int a = base.GetX();
    int b = base.GetY(); // NG 基本クラスにはGetYはない
    int c = derived.GetX(); // 基本クラスのGetXを継承
    int d = derived.GetY();
}
```

クラス継承とオーバーライド

基本クラス

```
class BaseClass2
{
private:
    int x;
public:
    int Get() {return x;}
    void Set(int ix) {x=ix;}
    void Raise() {
        x += 1;
    }
};
```

派生クラス

```
class DerivedClass2
    : public BaseClass2
{
public:
    // BaseClass::Raiseを上書き
    void Raise() {
        // x += 2; // NG
        Set(Get() + 2);
    }
};
```

クラス継承とオーバーライド contd.

```
int main()
{
    BaseClass2 base;
    DerivedClass2 derived;

    base.Set(10);    // baseとderivedに同じ値をセット
    derived.Set(10);

    base.Raise();    // BaseClass2::Raiseを呼び出し
    derived.Raise(); // DerivedClass2::Raiseを呼び出し
    int a = base.Get();
    int b = derived.Get();
    printf("base = %d, derived = %d\n", a, b); // ?
}
```


クラス継承とオーバーライド contd.

```
int raiseAndGet(BaseClass2& a) {  
    a.Raise();  
    return a.Get();  
}  
  
int main() {  
    BaseClass2 base;  
    DerivedClass2 derived;  
  
    base.Set(0);    // baseとderivedに同じ値をセット  
    derived.Set(0);  
    int a = raiseAndGet(base);  
    int b = raiseAndGet(derived);  
    printf("base = %d, derived = %d\n", a, b); // ?  
}
```

仮想関数とオーバーライド

基本クラス

```
class BaseClass3
{
private:
    int x;
public:
    int Get() {return x;}
    void Set(int ix) {x=ix;}
    virtual void Raise() {
        x += 1;
    } // 仮想関数と呼ばれる
};
```

派生クラス

```
class DerivedClass3
    : public BaseClass3
{
public:
    // Raiseをオーバーライド
    void Raise() override {
        Set(Get() + 2);
    }
};
```

仮想関数とオーバーライドcontd.

```
int raiseAndGet(BaseClass3& a) {  
    // ポリモーフィズム(多態性)  
    a.Raise(); // 引数aの実際のクラスに応じて呼び出し先が変わる  
    return a.Get();  
}  
  
int main() {  
    BaseClass3 base;  
    DerivedClass3 derived;  
  
    base.Set(0);    // baseとderivedに同じ値をセット  
    derived.Set(0);  
    int a = raiseAndGet(base);    // ?  
    int b = raiseAndGet(derived); // ?  
    printf("base = %d, derived = %d\n", a, b);  
}
```

本日の内容

- クラスの宣言と定義
 - ヘッダファイル(.h)とソースファイル(.cpp)
- 演習1
- 純粹仮想関数
- 抽象クラス, インターフェース
- 基底クラスメンバ関数の呼び出し
- 演習2

クラスの宣言 & 定義

main.cpp

```
class Vector2
{
public:
    Vector2(double ix, double iy) {
        x = ix;
        y = iy;
    }
public:
    double Length() const {
        return sqrt(x*x + y*y);
    }
}
```

```
void Print() const {
    cout << x << ", " << y << endl;
}

public:
    double x;
    double y;
};

int main()
{
    Vector2 u(1.0, 1.0);
    u.Print();
}
```

クラスの宣言 + 定義

ヘッダファイル : Vector2.h

```
class Vector2
{
public:
    Vector2(double ix, double iy);
public:
    double Length() const;
    void Print() const;
public:
    double x;
    double y;
};
```

メンバ変数とメンバ関数の
宣言のみ
= クラスの設計/骨組み
を利用者に公開

ソースファイル : Vector2.cpp

```
#include "Vector2.h" ヘッダファイルを含める
Vector2::Vector2(double ix, double iy)
{
    x = ix;
    y = iy;
}
double Vector2::Length() const
{
    return sqrt(x*x + y * y);
}
void Vector2::Print() const
{
    cout << x << ", " << y << endl;
}
```

メンバ関数の定義を記述
= クラスの内部動作を実装

なぜ分離する？

- 複数のソースファイルから利用したいクラス
 - iostreamやcmathなどの汎用的機能群
 - 「インターフェース」だけ公開, 内部動作は知らなくていい
- ヘッダファイルに宣言と定義を書いても良い
 - header-only library (Boost, Eigen)
 - ヘッダファイルが長くなって読みづらい
 - コンパイル時間が長くなる
 - 特に大規模開発では無視できない(回避する仕組みはある)
- Java, C#, python などに慣れていると意味不明
 - 同じようなコードを2回も書くのは確かに面倒
 - 言語仕様上の意味を知りたい人は独習を

ヘッダファイルとソースファイル演習

- header/main.cpp内に実装されている
Rectangleクラスについて，動作を保ちつつ
 - 宣言をRectangle.hに
 - 定義をRectangle.cppにそれぞれ分割して記述する

※リファクタリング

純粹仮想関数

```
class Shape
```

```
{
```

```
    virtual double Area() const = 0;
```

```
    virtual double Circumference() = 0;
```

```
public:
```

```
    Shape();
```

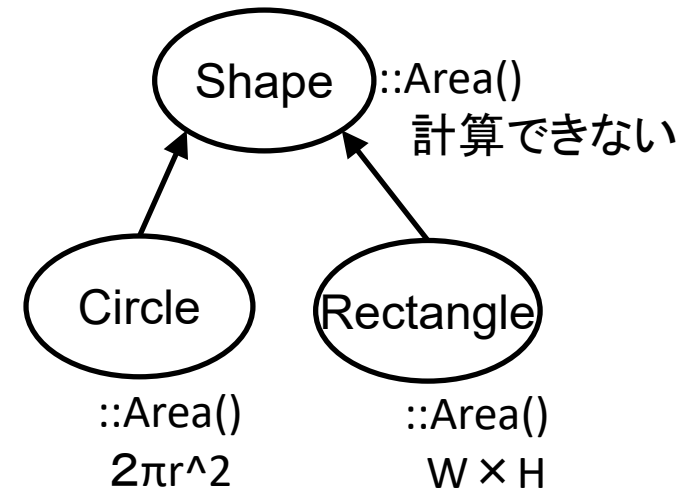
```
    ....
```

```
};
```

Shapeクラスでは定義
されない仮想関数



Shapeクラスを継承したら
オーバーライドして動作を
定義しなければならない



抽象クラス：純粹仮想関数を一部含む

ヘッダファイル：Shape.h

```
class Shape
{
public:
    virtual double Area() const = 0;
    virtual double Circumference() const = 0;
public:
    Shape(int ix, int iy); // Shape.cppで定義
private:
    double x;
    double y;
};
```

main.cpp

```
#include "Shape.h"

int main() {
    Shape shape(0, 0); // エラー
    shape.Area();      // エラー
}
```

抽象クラスの継承

Rectangle.h

```
#include "Shape.h"
class Rectangle : public Shape
{
public:
    double Area() const override;
    double Circumference() const override;
public:
    Rectangle(double ix, double iy,
               double iw, double ih);
private:
    double w, h;
}
```

Rectangle.cpp

```
#include "Rectangle.h"
double Rectangle::Area() const {
    return w * h;
}
double Rectangle::Circumference() const {
    return ???;
}
Rectangle::Rectangle(double ix, double iy,
                     double iw, double ih)
: Shape(ix, iy) {
    w = iw; h = ih;
}
```

Shapeクラスのコンストラクタを呼び出して、中心位置を設定

純粹仮想関数を使う理由

- 派生先ごとに挙動が異なるが、意味的には同じ機能に同一名称を与え、各派生クラスで定義させるよう強制できる
- 例：
 - 「図形には面積がある」「図形には周囲長が定義される」という条件を基底抽象クラスに記述
 - 各派生クラスで純粹仮想関数をオーバーライドし、「面積は $2\pi r^2$ で求める」「長方形の周囲長は全辺の長さの和によって求められる」という具体的計算方法を定義することで、上記条件を達成

ポインタを通じたポリモーフィズム

```
Shape *shapes[3]; //Shapeクラスへのポインタ
shapes[0] = new Rectangle;
shapes[1] = new Circle;
shapes[2] = new Star;
for (int i = 0; i < 3; ++i)
{
    // 図形に応じたArea計算を呼び出し
    std::cout << shape[i]->Area();
}
```

まとめ

- クラス, メンバ変数, メンバ関数(おさらい)
- アクセス指定子とカプセル化(おさらい)
- オーバーロードとオーバーライド(おさらい)
- 継承, 基底クラスと派生クラス(おさらい)
- 仮想関数と純粋仮想関数
- 抽象クラスとその継承

抽象クラスの継承演習

- Shapeフォルダ内の各ファイルを修正して、main.cpp に記述されているプログラムを正常に動作させる