

# 動的メモリ管理

Game Programming B #04

向井 智彦

# 先週のおさらい

- 値渡しと参照渡し
- ポインタ, メモリアドレス
- ポインタ変数の演算
- 配列とポインタ, ポインタ演算
- 特殊なポインタ `nullptr`, `this`
- ポインタを活用したデータ構造
  - リスト
  - ツリー

# 本日の内容

- 動的メモリ管理
  - new と delete, new[] と delete[]
  - コンストラクタとデストラクタ
- クラス継承とポインタ
  - protected アクセス
  - ポリモーフィズム revisited
  - アップキャストとダウンキャスト
- クラスインスタンスへのポインタの配列
- 演習：図形データ管理システムの開発

# 静的メモリ管理

- 何個の変数・配列を用いるのか, 全てソースコードに記述
  - プログラム動作中には変更できない

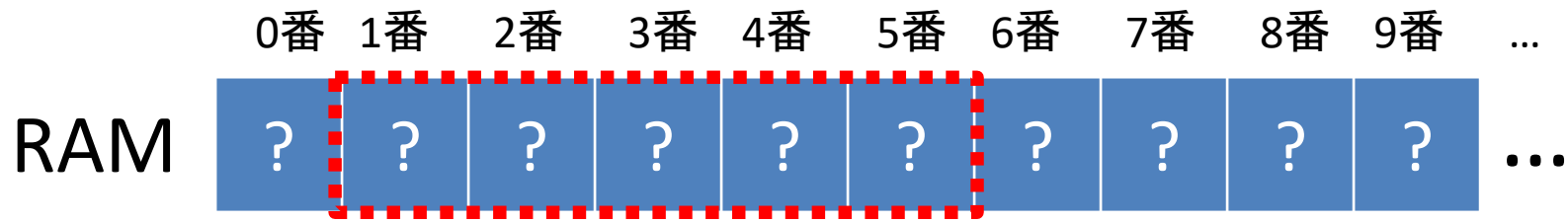
```
double x = 10;  
int a[6] = {1, 2, 3, 4, 5, 6};  
char c = 'A';
```



# 動的メモリ管理

- プログラム動作中に必要な変数・配列を確保したり, 不要な変数・配列を破棄する仕組み

```
int count;  
std::cin >> count;  
float *p = new float[count];  
delete[] p;
```



配列 p (長さはプログラム動作時まで不明)

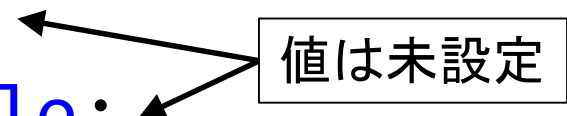
# 単一オブジェクトのnew & delete

変数へのポインタ型 ポインタ変数名

= new 変数型(コンストラクタ引数);

delete ポインタ変数名;

```
int      *pi = new int;
double   *pd = new double;
Vector2  *pv = new Vector2(0.0, 0.0);
Circle   *pc = new Circle(0, 0, 50);
delete pi;   delete pd;
delete pv;   delete pc;
```



値は未設定

# オブジェクト配列のnew[] & delete[]

変数へのポインタ型 配列名

= new 変数型[要素数];

delete[] ポインタ変数名;

```
int      *ai = new int[2];
double   *ad = new double[3]();
Vector2  *av = new Vector2[4];
Circle   *ac = new Circle[5]();
delete[] ai;   delete[] ad;
delete[] av;   delete[] ac;
```

値は未設定

ゼロクリア

デフォルト  
コンストラクタが  
呼び出される  
(Vector2::Vector2(),  
Circle::Circle())

# ポインタを通じたポリモーフィズム

```
Shape *shapes[3]; // Shapeクラスのポインタ配列
```

```
shapes[0] = new Rectangle;
```

```
shapes[1] = new Circle;
```

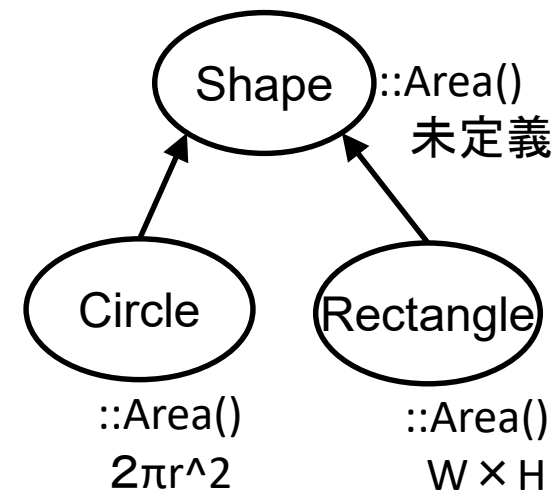
```
shapes[2] = new Star;
```

```
for (int i = 0; i < 3; ++i)
{
```

```
    // 図形に応じたArea計算を呼び出し
```

```
    std::cout << shapes[i]->Area();
```

```
}
```



(純粹) 仮想関数



# 全てを動的に...ポインタ配列

```
int n;      std::cin >> n;
Shape **shapes = new Shapes*[n]; //ポインタ配列
shapes[0] = new Rectangle;        //(Shape*) の*
....
shapes[n - 1] = new Circle;
for (int i = 0; i < n; ++i)
{
    // 図形に応じたArea計算を呼び出し
    std::cout << shapes[i]->Area() << std::endl;
    delete shapes[i];
}
delete[] shapes;
```

# protected アクセス

```
class Shape {
    double X() const;
    void SetX(double);
private:
    double x, y;
};
class MovableShape
: public Shape {
    void MoveRight() {
        SetX(X() + 1.0);
    }
};
```

派生クラスから直接アクセス  
できないのでメンバ関数経由

```
class Shape {
    double X() const;
    void SetX(double);
protected:
    double x, y;
};
class MovableShape
: public Shape {
    void MoveRight() {
        x += 1.0;
    }
};
```

派生クラスからも直接アクセス可  
クラス外からは依然不可  
(仮想関数オーバーライド時に便利)

# 基底クラスのポインタを通じた管理

```
Shape *shapes[2]; // Shapeクラスのポインタ配列  
shapes[0] = new Rectangle;  
shapes[1] = new Circle;  
cout << shapes[1]->Radius(); // エラー
```

基底クラスで未宣言の関数は使用不可

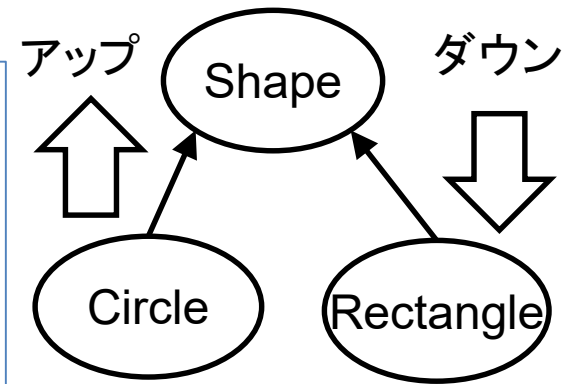
= 派生クラスで新たに宣言された関数は使用不可

しかし実際にはshapes[1]が指すのはCircleオブジェクト

# キャスト(型変換)

## アップキャスト

```
Shape *shapes[2];  
shapes[0] = new Rectangle;  
shapes[1] = new Circle;
```



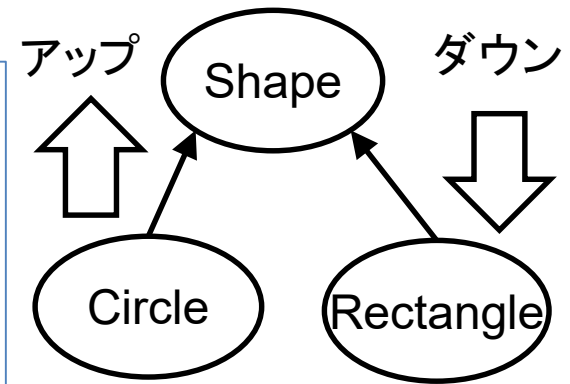
## ダウンキャスト

```
Rectangle *r =  
    dynamic_cast<Rectangle*>(shapes[0]);  
Circle *c =  
    dynamic_cast<Circle*>(shapes[0]);
```

# 危険なダウンキャスト

## アップキャスト

```
Shape *shapes[2];  
shapes[0] = new Rectangle;  
shapes[1] = new Circle;
```



ダウンキャスト : いずれも失敗して nullptr となる

```
Circle *c =  
    dynamic_cast<Circle*>(shapes[0]);  
Rectangle *r =  
    dynamic_cast<Rectangle*>(shapes[1]);
```

# クラスインスタンスへのポインタ配列： 派生クラスごとの管理

```
Rectangle *rect[2];  
Circle *circle[2];  
rect[0] = new Rectangle;  
circle[0] = new Circle;  
cout << rect[0]->Area() << endl;  
cout << circle[0]->Area() << endl;  
circle[0]->Radius();
```

- 派生クラスが少ない限りにおいては読みやすい
  - 派生数が一定数を超えた途端にメンテナンス困難

# クラスインスタンスへのポインタ配列： 基底クラスを通じた管理

```
Shape *shapes[2];  
shapes[0] = new Rectangle;  
shapes[1] = new Circle;  
cout << shapes[0]->Area() << endl;  
cout << shapes[1]->Area() << endl;  
dynamic_cast<Circle*>(shapes[1])->Radius();
```

- 基底クラスへのポインタ配列として管理
- 基底クラスに宣言された共通機能は容易に利用
  - 仮想関数のオーバーライドと組み合わせ
- 派生クラス固有の機能はダウンキャストを通じて利用

# クラス継承と仮想デストラクタ

```
Shape *shapes[2];  
shapes[0] = new Rectangle();  
shapes[1] = new Circle();  
cout << shapes[0]->Area() << endl;  
cout << shapes[1]->Area() << endl;  
delete shapes[0];  
delete shapes[1];
```

仮想デストラクタを用いないと

Shape::~~Shape が呼ばれる

(Rectangle::~~Rectangleや

Circle::~~Circle が呼び出されない)



# 基底クラスメンバ関数の呼び出し

```
class Data {  
    virtual void Hoge() {  
        何らかの処理40行  
    };  
class DataEx  
: public Data {  
    void Hoge() override {  
        追加の処理2行  
        何らかの処理40行  
    }  
};
```

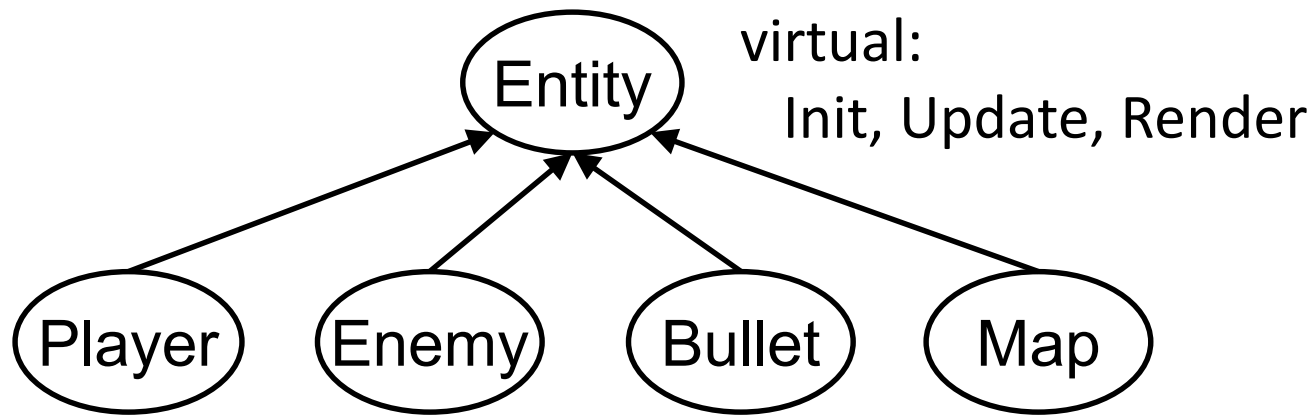
重複が無駄...

```
class Data {  
    virtual void Hoge() {  
        何らかの処理40行  
    };  
class DataEx  
: public Data {  
    void Hoge() override {  
        追加の処理2行  
        Data::Hoge();  
    }  
};
```

「基底クラス名::仮想関数名」で  
オーバーライド前の関数を利用可

# ゲームエンティティクラスの設計

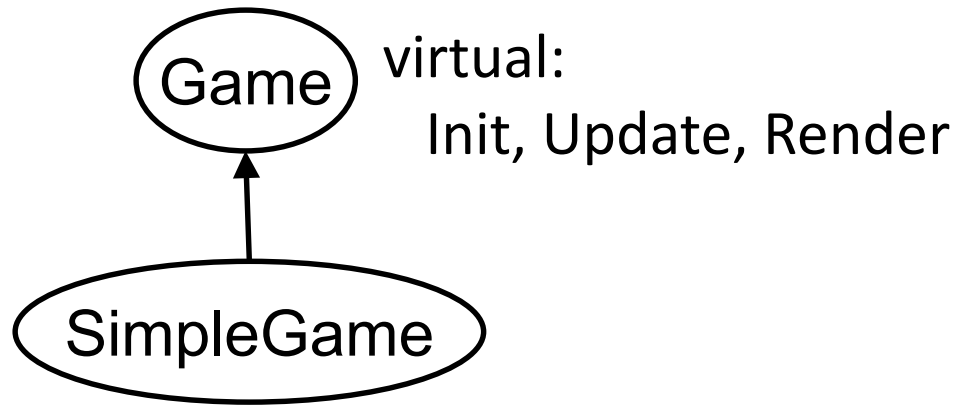
- ゲームに登場するモノ = エンティティ(Entity)
  - Entityクラスには全てのエンティティに共通する機能を宣言し, デフォルトの挙動を定義



- 各派生エンティティで, Entityのデフォルト機能をオーバーライド & 独自機能を追加

# ゲームクラス

- 多くのゲームに共通する機能を宣言し、それぞれデフォルトの挙動を定義



- 各派生ゲームで各ゲーム独自の動作をオーバーライド & 固有機能を追加

# 継承か？合成か？ is-a と has-a

- is-a: 「オリジナルゲーム」は「ゲーム」
- is-a: 「Playerエンティティ」は「エンティティ」
- is-a の場合はクラス継承を用いる
- has-a: 「ゲーム」は「エンティティ」を持つ
  - エンティティはゲームではなく、あくまで構成要素
- has-a の場合はクラスメンバ変数として合成
- 無意味な継承は避ける
  - 「has-a」ベースで設計できないか？と疑い続ける

# まとめ

- 動的メモリ管理
  - new と delete, new[] と delete[]
- クラス継承とポインタ
  - 基底クラスの仮想関数を通じたポリモーフィズム
  - アップキャストとダウンキャスト
    - ダウンキャストの扱いには注意
- クラスインスタンスへのポインタの配列
  - 基本的には基底クラスへのポインタを格納
- 継承と合成: is-a or has-a

# 説明しなかったこと

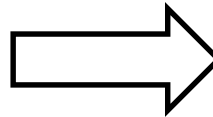
- スマートポインタ
  - `std::unique_ptr`, `std::shared_ptr`, `std::weak_ptr`
- その他のキャスト
  - `static_cast`, `reinterpret_cast`
- 継承の `public`, `protected`, `private`
- 静的メンバ変数, 静的メンバ関数
- フレンド

# 演習： 4種の図形データの管理

2次元図形データを6つ順番にキーボード入力すると、それら6つの図形の作成時サイズ情報、および面積、周囲長、重心位置、X方向の幅、Y方向の高さを出力

図形種類

```
>> 0 中心位置XY
>> 00 100 50 幅 & 高さ
>> 1
>> 10 0 50
...
>> 0
>> 10 10 50 100
```



```
Rectangle – (0, 0) – W100 x H50
Area = 50
Boundary length = 300
CoM = (0, 0)
X100 x Y50

Circle – (10, 0) – R50
Area = ...
```

# 演習の進め方(案)

1. 円と長方形以外の図形クラス2種を追加
  - それぞれ Shape クラスを継承し, 「図形種類番号」純粹仮想関数 Type()をオーバーライド
    - 四角形は 0, 円なら 1 を戻すような関数
  - 中心位置も適切に設定
  - 扱いやすい図形は三角形, 楕円, 平行四辺形?
    - コンストラクタの引数は要検討
      - 三角形なら3辺の長さ? 3角の角度? 頂点座標?
      - 二等辺三角形に限定するとラクできる



# 演習の進め方(案)

2. キーボード入力に応じて図形オブジェクトを6つ動的生成し、Shapeクラスポインタ配列に格納
  - 図形番号を指定後, 図形に応じた追加情報をキーボード入力
3. 出力時には、各派生クラスにダウンキャストして、それぞれの固有メンバ関数を利用
  - 図形番号 `Shape::Type()` に応じてダウンキャスト先クラスを決定
4. newしたオブジェクトを全てdelete