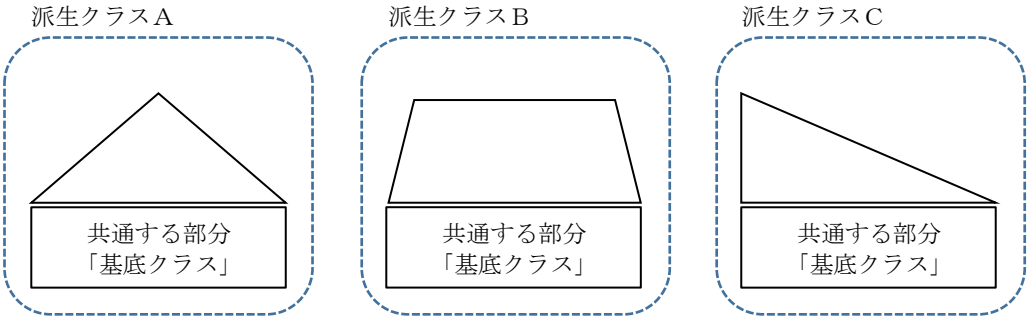
後期課題１

３日で分かるC++言語での基本プログラミング②

■継承

オブジェクト指向言語の重要な特性の一つに「継承（インヘリタンス）」があります。複数のオブジェクトを作る際に共通する部分が出て来た場合、その共通部分を基本のクラスとして設定しておき、実際のそれぞれのクラスは基本クラスの性質に独自の機能を拡張していくイメージになります。

その時の継承のもととなるクラスを「基底クラス（スーパークラス）」と言い。その基底クラスを継承して独自の機能を実装したクラスを「派生クラス（サブクラス）」と言います。



定義の仕方は、派生クラスを定義する時にベースとなる基底クラスを指定する形になります。

```
class 派生クラス名 : public 基底クラス名
```

CHARA_BASE.h　※ベースになるクラスは大文字などで表現すると分かり易い

```
class CHARA_BASE
{
public:
    int pos_x;
    int pos_y;
};
```

Player.h

```
class Player : public CHARA_BASE
{

};
```

Enemy.h

```
class Enemy : public CHARA_BASE
{

};
```

CHARA_BASEクラスを規定クラスとして、それぞれPlayerとEnemyの派生クラスを作成します。
このサンプルの場合は、Player、Enemyのクラスから基底クラスで定義した変数pos_x,posYが使用できます。

■オーバーロード

ポリモーフィズムの一つで、コンストラクタを含め、引数や戻り値が違うものであれば「同じ名称で全く違う複数のメンバ関数を定義する事ができる」というものです。引数がない場合の初期化や、引数を渡す場合の初期化など、用途に合わせた処理を同じ名称の関数で実行できる便利な機能となります。

Enemy.h

```
class Enemy : public CHARA_BASE
{
public:
   int x;
   int y;
   int type;

   Enemy();                 // ①引数のないｺﾝｽﾄﾗｸﾀ(関数)
   Enemy(int, int);         // ②引数が 2 個のｺﾝｽﾄﾗｸﾀ(関数)
   Enemy(int, int, int);    // ③引数が 3 個のｺﾝｽﾄﾗｸﾀ(関数)
};
```

Enemy.cpp

```
Enemy::Enemy()
{
    x = 0;
    y = 0;
    type = 0;
}

Enemy::Enemy(int a, int b)
{
    x = a;
    y = b;
    type = 0;
}

Enemy::Enemy(int a, int b, int t)
{
    x = a;
    y = b;
    type = t;
}
```

main.cpp

```
void main()
{
    Enemy* e;

    // ①   e = new Enemy();
    // ②   e = new Enemy(100, 200);
    // ③   e = new Enemy(100, 200, 5);
}
```

プログラム例の様に、オブジェクト生成する場合のコンストラクタに引数の渡し方を変えると、その引数に応じたメンバ関数が呼ばれる様になり、それぞれの処理を実行させる事ができます。（初期化やパラメータ数に応じた機能分けなどをする事ができます）。

尚、引数に何もないコンストラクタの事を「デフォルトコンストラクタ」と言います。

■オーバーライド

こちらもポリモーフィズムの一つで、親子関係にあるクラスにおいて規定クラスと派生クラスに同名・同型の関数がある場合、派生クラスの関数が規定クラスの関数を上書きするイメージになります。

CHARA_BASE.h

```
class CHARA_BASE
{
public:
    void Init(){
        // 処理①
    }
};
```

Player.h

```
class Player : public CHARA_BASE
{
public:
    void Init(){
        // 処理②
    }
};
```

main.cpp

```
void main()
{
    CHARA_BASE* c = new CHARA_BASE();
    Player* p = new Player();

    c->Init();     // 処理①を実行
    p->Init();     // 処理②を実行

};
```

オーバーライドされた関数は**原則的に派生クラスの方の関数を実行**します。

注意点！
- メンバ変数の名前を統一して記述ミスを減らす。
- 原則的に同じ機能には同じ名前を付けて処理に統一感を持たせる。

□派生クラスのコンストラクタとデストラクタの順番は？

オブジェクトが生成されるとコンストラクタが呼ばれ、オブジェクトが削除されるとデストラクタが呼ばれますが、親クラスのコンストラクタとデストラクタの呼ばれるタイミングはどうなるのでしょう？

順番としては

・コンストラクタ・・・・・基底クラスのコンストラクタ　→　派生クラスのコンストラクタ

・デストラクタ・・・・・・派生クラスのデストラクタ　→　基底クラスのデストラクタ

となります。

が、デストラクタには一部注意点があります。

```
class BaseClass
{
public:
    BaseClass(){  // ①基礎ｸﾗｽのｺﾝｽﾄﾗｸﾀ  }
    ~BaseClass(){  // ②基底ｸﾗｽのﾃﾞｽﾄﾗｸﾀ  }
};

class SubClass : public BaseClass
{
     SubClass(){  // ③派生ｸﾗｽのｺﾝｽﾄﾗｸﾀ  }
     ~SubClass(){  // ④派生ｸﾗｽのﾃﾞｽﾄﾗｸﾀ  }
};


void main()
{
    SubClass* sub = new SubClass();  // Sub ｸﾗｽのﾎﾟｲﾝﾀに Sub ｸﾗｽのﾎﾟｲﾝﾀを代入
    delete sub;

    BaseClass* base = new SubClass();  // Base ｸﾗｽのﾎﾟｲﾝﾀに Sub ｸﾗｽのﾎﾟｲﾝﾀを代入
    delete base;
}
```

実行時のコンストラクタ・デストラクタの動き

```
----- SubClass のポインタに Sub クラスを構築
①基底クラスのコンストラクタを実行
③派生クラスのコンストラクタを実行
④派生クラスのデストラクタを実行
②基底クラスのデストラクタを実行

----- BaseClass のポインタに Sub クラスを構築
①基底クラスのコンストラクタを実行
③派生クラスのコンストラクタを実行
※派生クラスのデストラクタが呼ばれない
②基底クラスのデストラクタを実行
```

2回目の例では、基底クラスのポインタに派生クラスをインスタンスしているので、デストラクタ時の「自身」とは「基底クラス」になっています。その為、派生クラスのデストラクタが呼ばれなくなってしまいます。

ですので、そうならない為にも、継承（ポリモーフィズム）をする場合は「基底クラスのデストラクタを **virtual(バーチャル)修飾子**」にして、デストラクタ時に派生クラスが呼ばれる様に「**基底クラスの関数を仮想にしておく**」と良いです。

```
class BaseClass
{
public:
    BaseClass(){  // ①基底ｸﾗｽのｺﾝｽﾄﾗｸﾀ  }
    virtual ~BaseClass(){  // ②基底ｸﾗｽのﾃﾞｽﾄﾗｸﾀ  }
};

class SubClass : public BaseClass
{
     SubClass(){  // ③派生ｸﾗｽのｺﾝｽﾄﾗｸﾀ  }
     ~SubClass(){  // ④派生ｸﾗｽのﾃﾞｽﾄﾗｸﾀ  }
};


void main()
{
    SubClass* sub = new SubClass();  // Sub ｸﾗｽのﾎﾟｲﾝﾀに Sub ｸﾗｽのﾎﾟｲﾝﾀを代入
    delete sub;

    BaseClass* base = new SubClass();  // Base ｸﾗｽのﾎﾟｲﾝﾀに Sub ｸﾗｽのﾎﾟｲﾝﾀを代入
    delete base;
}
```

実行時のコンストラクタ・デストラクタの動き

```
----- SubClass のポインタに Sub クラスを構築
①基底クラスのコンストラクタを実行
③派生クラスのコンストラクタを実行
④派生クラスのデストラクタを実行
②基底クラスのデストラクタを実行

----- BaseClass のポインタに Sub クラスを構築
①基底クラスのコンストラクタを実行
③派生クラスのコンストラクタを実行
④派生クラスのデストラクタを実行
②基底クラスのデストラクタを実行
```

■virtual（仮想関数）

仮想関数とも呼ばれ、継承したクラスで同名の関数を作った際、基底の関数に virtual キーワードを付けておくと、継承したオブジェクトを基底クラスとして扱っても、その際に呼び出される関数は派生クラスの関数が呼び出される様になるというものです。

文字だと分かり辛いので図にしてみます。

virtual 関数の有無の違い

```
class BASE
{
public:
    void Action(){ ①を実行 }
};

class Player : public BASE
{
public:
     void Action(){ ②を実行 }
};


void main()
{
    BASE* p = new Player();
    p->Action();  // ①が実行される

    Player* p = new Player();
    p->Action();  // ②が実行される
}
```

```
class BASE
{
public:
    virtual void Action(){ ①を実行 }
};

class Player : public BASE
{
public:
     void Action(){ ②を実行 }
};


void main()
{
    BASE* p = new Player();
    p->Action();  // ②が実行される

    Player* p = new Player();
    p->Action();  // ②が実行される
}
```

最初の頃は違いが分かり辛いと思いますが、今後「ポリモーフィズム」や「参照渡し」などが登場するにつれ、基底クラスと派生クラスの明確な使い分けを行う必要が出てきますので重要度が増してくると思います。

□純粋仮想関数

「継承先で必ず実装する関数なので基底オブジェクトでは定義しなくてよい（定義しない）」と、始めからはっきりしているものは、基底オブジェクトの virtual 定義時に、純粋仮想関数として定義しておきます。この関数が「一つでもあるクラス」は、純粋仮想クラスと言い、関数の中身がない為、単品でのオブジェクト化ができなくなります。

ですので、誰かに継承して貰うだけの役割りとしてのクラスを作る事になります。

```
virtual　戻り値　関数名（引数の型） = 0;
```

仮想関数

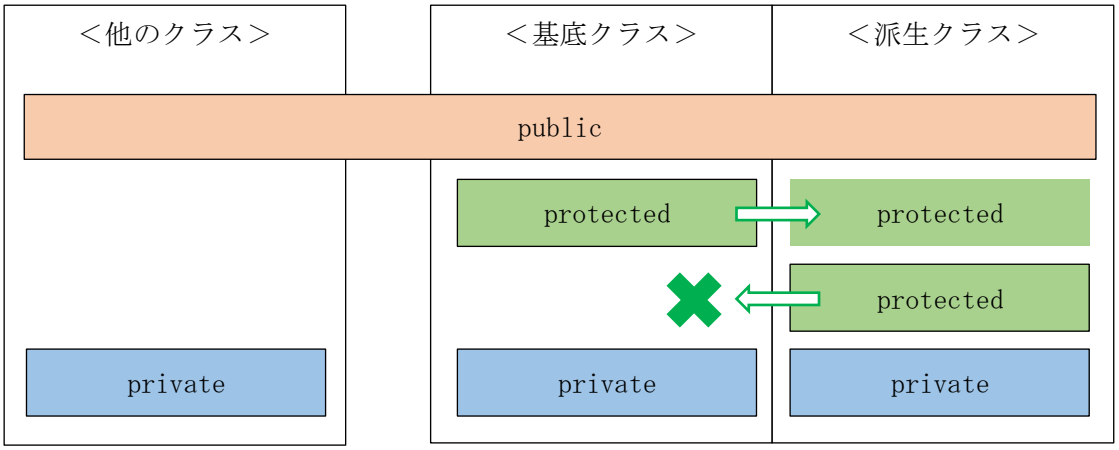
純粋仮想関数

■アクセス指定子 part.2

継承が出て来たので、アクセス指定子の残りの protected について説明していきます。

アクセス指定子	意味
public	全ての範囲からアクセス（読み込み・書き込み）可能
private	同一クラス、インスタンス内でのみアクセス可能
protected	同一クラス、インスタンスに加え、継承された派生クラスでもアクセス可能

protected メンバは private メンバ同様、クラス外からのアクセスはできません。private との違いは継承した派生クラスからは public 同様に扱えるという事で、派生クラスのみにアクセスをさせたい時に protected を付けます。
逆に派生クラスで protected で定義したものは基底クラスではアクセスできません。継承した先で有効となります。



■まとめ

このドキュメントで出て来た用語の一覧です。各キーワードの理解はいかがでしょうか。

・継承
・オーバーロード
・オーバーライド
・virtual（仮想関数）
・純粋仮想関数
・アクセス指定子 part.2（protected）