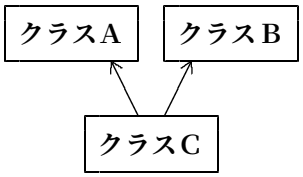
オブジェクト指向とゲームプログラミング

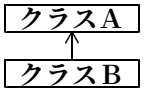
○　C++編 － 第８回　多重継承　○

多重継承

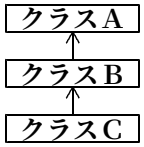
２つ以上のクラスを継承して派生クラスを作成することができます。このように、複数の基底クラスがある継承を多重継承と呼びます。多重継承に対し、基底クラスがひとつだけの継承を単一継承と呼びます。



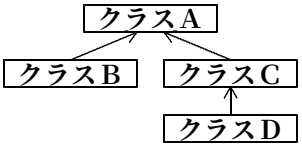
▲多重継承



▲単一継承



▲階層的な単一継承



▲単一継承

多重継承は、以下のような構文で行います。

```
class 派生クラス名 : アクセス指定子 基底クラス名, アクセス指定子 基底クラス名, … {
    追加・変更のメンバを記述
};
```

以下のプログラムでは、クラスＡとクラスＢを多重継承してクラスＣを作成しています。

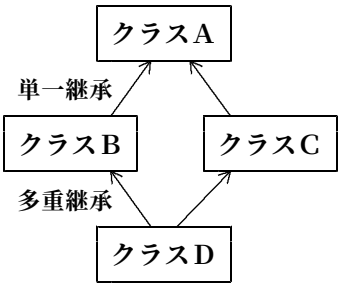
```
// クラスA
class A {
protected:
    int   a_;
};

// クラスB
class B {
protected:
    int   b_;
};

// クラスC
class C : public A, public B {
public:
    int GetValA() const { return a_; }
    int GetValB() const { return b_; }
};
```

基底クラスのアクセス指定が同じ場合は、省略することもできます。

```
// クラスC
class C : public A, B {
public:
    int GetValA() const { return a_; }
    int GetValB() const { return b_; }
};
```

多重継承の問題点

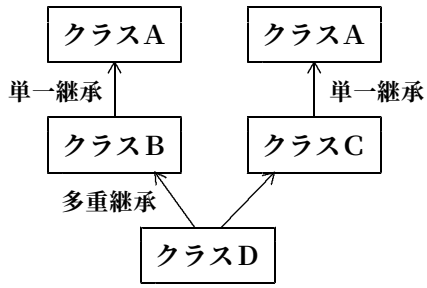
多重継承では、次のようになることがあります。あるクラスＡを継承してクラスＢとクラスＣを作成します。さらに、クラスＢとクラスＣを多重継承してクラスＤを作成します。



この継承は上図のようになります。ひし形のように継承が行われることから、ひし形継承またはダイアモンド継承と呼ばれます。このひし形継承を用いた以下のプログラムをコンパイルしてみると、エラーになってしまいます。

```
// クラスA
class A {
protected:
    int   a_;
};

// クラスB
class B : public A {};

// クラスC
class C : public A {};

// クラスD
class D : public B, public C {
public:
    void Init(const int arg) { a_ = arg; }  // エラー
}
```

「a_ = arg;」の部分がエラーになります。具体的には、メンバ変数a_は、クラスBのものなのか、それともクラスCのものなのかがあいまいである、というエラーメッセージが表示され、コンパイルが中止されます。前掲図では、クラスAが１つに見えますが、実際には以下のように継承されているものとみなされます。



そのため、それぞれ共通した基底クラスのメンバが複数存在することになります。この例では、クラスB専用の基底クラスAと、クラスC専用の基底クラスAが、それぞれ影響を及ぼすことなく独立して存在するのです。このようなあいまいさを解決するため、スコープ解決演算子::を使って次のようにすることでエラーを回避することはできます。

```
// クラスD
class D : public B, public C {
public:
    void Init(const int arg) { B::a_ = arg; C::a_ = 0; }    // エラーにならない
}
```

しかし、クラスAのすべてのメンバに対してスコープ解決演算子が必要になってしまいます。また、本来ひとつでよいはずのクラスAが複数存在してしまうという問題も残ったままです。

仮想継承

C++では、「多重継承したとき同じ基底クラスが複数存在する場合がある」という問題を解決するため、仮想継承という概念が用意されています。仮想継承を行うと、そのクラスはインスタンス内で共有されます。仮想継承された基底クラスを仮想基底クラスと呼びます。仮想基底クラスを多重継承して派生クラスを複数作成しても、仮想継承されたクラスはただひとつだけであることが保証されるのです。
仮想継承を行うには、キーワードvirutalをつけて継承します。

```
// クラスA
class A {
protected:
    int   a_;
};

// クラスB
class B : virtual public A {};  // 仮想継承。クラスAは仮想基底クラスとなる

// クラスC
class C : virtual public A {};  // 仮想継承。クラスAは仮想基底クラスとなる

// クラスD
class D : public B, public C {
public:
    void Init (const int arg) { a_ = arg; }                 // エラーにならない
    void Init2(const int arg) { B::a_ = arg; C::a_ = 0; }
    // ↑の場合でも、共有されたクラスAを指すので、最後に実行された方の値になる。
    // すなわち、クラスBのa_もクラスCのa_も0になる。
};
```

この場合、クラスAは仮想継承されています。クラスAは仮想基底クラスとなり、どんなに多重継承されてもそのクラス内にひとつしか存在しないことが保証されます。つまり、クラスBの基底クラスAとクラスCの基底クラスAは、共有されてどちらも同じ領域となります。 クラスDの中にクラスAはひとつしか存在しないので、「B::a_」などとせずに「a_」と記述できるようになります。また、共有されているので「B::a_」の値を変更すると「C::a_」の値も同じものに変更されます。

仮想継承とコンストラクタ

仮想継承した場合でも、コンストラクタは基底クラスから順に実行されますが、仮想基底クラスのコンストラクタは一度しか実行されないので、以下のようになります。

1. 仮想継承されたクラスのコンストラクタ(仮想基底クラスのコンストラクタ)。仮想基底クラスが複数ある場合は宣言順
2. 非仮想基底クラスのコンストラクタ。非仮想基底クラスが複数ある場合は宣言順
3. 派生クラスのコンストラクタ

仮想基底クラスの引数付きコンストラクタを呼び出すには、最後に派生したクラスのコンストラクタ初期化子を用いるしかありません。以下のようなクラスを作成したとします。

```
// クラスA
class A {
protected:
    A() : a_(0) {}                  // デフォルトコンストラクタ
    A(const int arg) : a_(arg) {}   // コンストラクタ

    int   a_;
};

// クラスB
class B : virtual public A {
protected:
    B(const int arg) : A(arg) {}    // コンストラクタ
};

// クラスC
class C : virtual public A {
protected:
```

```
    C(const int arg) : A(arg) {}    // コンストラクタ
};

// クラスD
class D : public B, C {
public:
    // コンストラクタ
    D(const int arg1, const int arg2) : B(arg1), C(arg2)
    {
        // コンストラクタ内処理
    }
};
```

クラスDのオブジェクトを以下のように生成します。

```
D   d(2, 3);        // オブジェクト生成後、a_の値は？
```

オブジェクトdが生成されるとき、次のような動作になります。

①クラスDの引数付きコンストラクタが呼び出される
②仮想基底クラス(クラスA)のコンストラクタを呼び出す。クラスDには、クラスAの明示的なコンストラクタ初期化子がないので、デフォルトコンストラクタが呼ばれる。
③非仮想基底クラス(クラスBおよびC)のコンストラクタを呼び出す。クラスDには、クラスBおよびCの明示的なコンストラクタ初期化子があるので、宣言順に処理する(B→Cの順)
④クラスDのコンストラクタ内の処理を行う

この順番からすると、生成されたオブジェクトdのメンバ変数a_は「3」になりそうですが、実際には「0」になります。

クラスAは、②の動作により、③の動作より先に生成されています。クラスBおよびCのコンストラクト時点では、すでに生成されているのです。そのため、クラスBおよびCのコンストラクタ初期化子「A(arg)」は実行されません。コンストラクタは、インスタンスの生成時に呼び出されるのであり、すでに生成されている場合には呼び出されないからです。つまり、クラスBおよびCが呼び出しているクラスAを生成するためのコンストラクタが無視されるのです。クラスBやCからすれば、自分の指定したとおりに初期化されない可能性があるわけです(ただし、コンストラクタ初期化子を用いず、コンストラクタ内でa_を初期化した場合は無視されません)。

結果として、最初に実行されたクラスAのコンストラクタ、すなわちクラスDが暗黙のうちに呼び出しているクラスAのデフォルトコンストラクタだけが有効となり、「a_」の値は0になります。

仮想継承と仮想関数

仮想継承と仮想関数を同時に用いる場合があります。このとき、以下のように、最後に派生したクラスでオーバーライドがなされている場合は、問題なくコンパイルできます。

```
// クラスA
class A {
public:
    virtual int GetVal() const { return 10; }
};

// クラスB
class B : virtual public A {
public:
    virtual int GetVal() const { return 20; }   // オーバーライド
};

// クラスC
class C : virtual public A {
public:
    virtual int GetVal() const { return 30; }   // オーバーライド
};

// クラスD
class D : public B, C {
public:
```

```
    virtual int GetVal() const { return 40; }   // オーバーライド
};
```

ところが、最後に派生したクラスでオーバーライドされていない場合は、エラーまたは警告になる場合があります。クラスDを以下のようにすると、クラスDがまったく同じ性質のクラスBのメンバ関数GetValとクラスCのメンバ関数GetValを持つことになり、コンパイルできません。

```
// クラスD
class D : public B, C {
public:
    // いずれの関数もオーバーライドしない
};
```

この状態から、クラスC(またはクラスB)でオーバーライドしないようにすると、コンパイルできるようになります。

```
// クラスC
class C : virtual public A {
public:
    // いずれの関数もオーバーライドしない
};
```

しかし、コンパイルはできますが、結局クラスDでは、クラスBが継承したGetVal関数と、クラスCがクラスAから継承したGetVal関数の２つ存在することになり､コンパイラによっては警告となります。このまま以下のプログラムを実行すると、どうなるのでしょうか。

```
A*   d = new D;
const int   ret = d->GetVal();
```

変数retには、「２０」が代入されます。これは、最後にオーバーライドされたクラスBの仮想関数が、クラスCがクラスAからそのまま継承した非オーバーライド関数より優先されるためです。なお、以下のようにスコープ解決演算子でクラスを明示した場合は、あいまいさがまったくないので、エラーにも警告にもならず、そのクラスの仮想関数が呼び出されます。

```
A*   d = new D;
const int   ret = d->A::GetVal();   // ret = 10
```

練習問題

1　ひし形継承の問題点とその解決策について述べましょう。

2　仮想継承した場合のコンストラクタの呼び出しについて、注意すべき点を述べましょう。

3　仮想継承を用いて多重継承した場合の仮想関数について、注意すべき点を述べましょう。