「継承」で説明した通り、C#ではクラスの多重継承を認めていません。すなわち、どんな型も基底クラスは1つだけです。制限なしの多重継承は実装のコストが結構高く、メリットに対してコストが見合わないというのが一般的な見方です。ここでは、その多重継承にかかるコストについて説明していきます。

# オブジェクトのレイアウト

メモリレイアウトの話、どこかで書いたはずなのでリンクで済ます。

おさらい

class A

{

int x;

int y;

int z;

}

があったときにメモリ レイアウトがどうなってるか。

# 継承時のレイアウト

続いて、クラスを継承したときのレイアウトがどうなるかを説明します。例えば、以下のようなクラス階層を考えます。

class A

{

int x;

}

class B : A

{

int y;

}

class C : B

{

int z;

}

このとき、A, B, Cのインスタンスは、それぞれ以下のようなレイアウトになります。

A B C

x x x

y y

z

継承の性質(is-aの関係)から、BのインスタンスはAとしても使え、CのインスタンスはBとしてもAとしても使えます。単一継承の場合、実装上も難しい話は何もなく、単に<em>前半だけを使う</em>ことで基底クラスとして振る舞うことができます。

図

Bのxのところだけ囲って、ここだけ見ればA

Cのxのところ同上、x,yのところだけ見ればB

# 多重継承時のレイアウト

面倒が出てくるのはここからです。先ほどと同じことを多重継承で考えてみましょう。

C#では認められていませんが、仮に、以下のように書けたとしましょう。

class A

{

int x;

}

class B

{

int y;

}

class C : A, B

{

int z;

}

レイアウトは以下のようにするのが一番シンプルです。

A B C

x x x

y

z

Cでは、x(Aのメンバー)とy(Bのメンバー)のどちらを前にするかという問題はあるものの、まあ、宣言の順(C : A, Bという順で書いたんだからAが先)にすればいいでしょう。ここで、CはAでもBでもあります。CのインスタンスをAとして使うのは「前半を使う」で問題ないんですが、Bとして使いたければ「真ん中を使う」という特殊ルールが必要になります(先頭からずれている分のオフセット管理が必要)。

図

「真ん中を使う」というのは単純そうにも思えますが、実装上は結構面倒になります。BのインスタンスとCのインスタンスを混在させて、どちらもBとして使うことを考えると、Bのインスタンスなら普通に先頭を、Cのインスタンスなら真ん中を使うというような処理が必要になります。

# ダイヤモンド継承階層

多重継承を認めた場合には特に面倒になるのはダイヤモンド継承と呼ばれる継承を行った場合です。以下のような継承を考えてみましょう。

class A

{

int x;

}

class B : A

{

int y1;

}

class C : A

{

int y2;

}

class D : B, C

{

int z;

}

階層を絵にすると、以下のようにひし形になることからダイヤモンド継承と呼ばれます。

図にする

では、これらのクラスのレイアウトを考えてみましょう。A, B, Cについては単一継承しかしていないので問題はありません。

図にする

一方で、Dクラスのレイアウトをどうすべきはなかなか悩ましくなります。BにもCにも、Aのメンバーxが含まれていることが問題になります。Dからすると、二重管理になります。

よくある実装としては2パターンあります。1つ目は本当に二重管理してしまう方法xを2か所に持ってしまいます。

図にする

x,y1のところをAとして、x,y2のところをB4して使う

このやり方だと、DのインスタンスをCとして使うのに、前節と同じ「真ん中を使う」(オフセット管理)でできるという利点はあります。一方で、やはり二重管理は大変です。xを書き換えるたびに、2か所のメモリ上を書き換える必要があります。

二重管理をやめるなら、以下のレイアウトが考えられるでしょう。

図

x

y1

y2

z

今度の問題は、DのインスタンスをCとして使う場合です。Cのメンバーがとびとびになっています。

図

つまり、以下のような特殊なオフセット管理が必要になります。

図

x y２

元からCのインスタンス 0 1

DをCとして使う 0 2

こういう特殊なオフセット管理はコストに直結します。フィールドの読み書きがだいぶ遅くなるわけです。そのコストを掛けてまで、本当に多重継承を使いたかったのかということを考えないといけません。

実際にほしかったものはインターフェイス

ちなみに、実際、C++の場合はここで説明したようなメモリ レイアウトで多重継承を実装しています。しかし、コストが高すぎるので避けられる傾向があります。

ただし、他重継承のコストが問題にならない場合が1つあります。それは、何もフィールドを持っていない場合です。フィールドのメモリ レイアウトが問題なんだから、フィールドがなければ問題は起きません。フィールドがなくても、メソッドがあればクラスの振る舞いは定義できます。フィールドなしだとメソッドの実装が書けないにしても、抽象メソッドであればそもそも実装は必要ありません。

つまり、抽象メソッドだけを持つクラス(C++的に言うと純粋仮想関数だけを持つクラス)なら多重継承しても無害です。C++でもそういうクラスを作ることが推奨されていますし、それを言語構文として取り入れたのがJavaやC#のインターフェイスです。なので、JavaやC#では、クラスは単一継承関係しか認めず、代わりに、インターフェイスの実装が無制限です。

もちろん、フィールドを持てない分の不便はありますが、抽象メソッドだけでも十分な利用価値があります。コストとメリットのバランスを考えると、これが良い妥協点でした。

mix-in

•mix-in的な話はあるものの、それを継承でやるのは無理筋 ◦mix-inはhas-aでやるべき、mix-in用の構文があるべき