メンバとアクセス制御

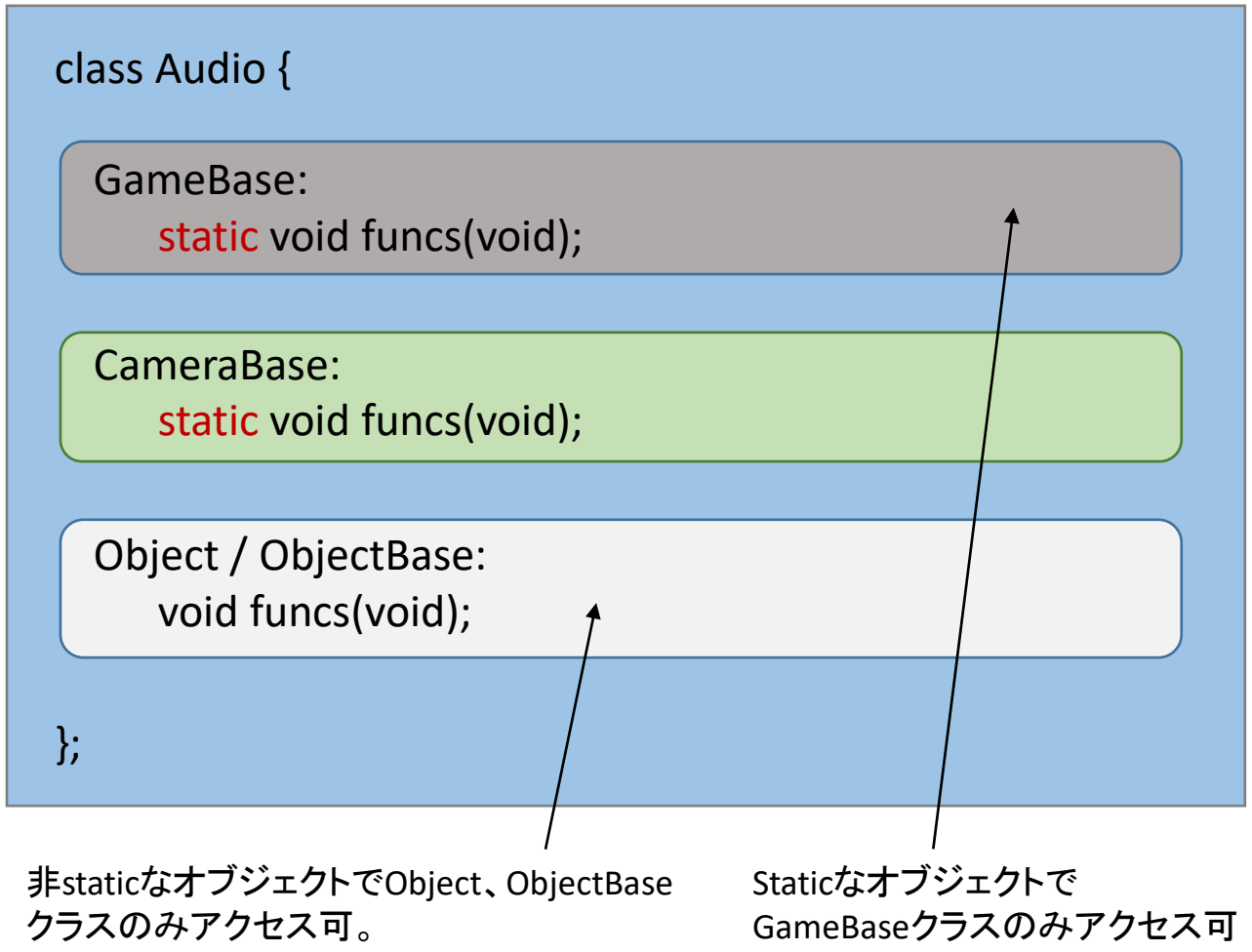
Atsuhito Fukuda

[概要]

My Game Programming in C++では、

「各々にとって必要な箇所のみにアクセスできるようなclass宣言を書きたい」

という考えから右のようなclass宣言をイメージしてヘッダを作成しています。



順を追ってこれを説明していきます

[C++で相手に応じて必要なメンバだけにアクセスさせたい]

右のようなclass Aをヘッダに宣言するとします。関数Update()はクラスBに、関数Initialize()はクラスCにとって必要となりますが、双方がpublicなメンバ関数であるためにアクセスするべきでないメンバにも両クラスはアクセスできてしまいます。

どのクラスがどの関数を用いてよいか注意書きをすることもできますが、それではチェック作業を増やしプログラマの手間になるのではないかと思います。

誤用の可能性をなくすためには不要なアクセスがそもそもできないことが望ましいはずですし、使用先を確実に制限できれば管理も易しくなると思います。

(A.h)

```
class A{

 public:
    static void Update(void);   // -> Bクラス
    static void Initialize(void);   //-> Cクラス

};
```

[friendを用いたアクセス制御 (staticメンバの場合)]

My Game Programming in C++では頂点データをファイルから配列に読み込む等の処理を扱う(object::data::)DataBaseクラスを用意しています。(object::data, object::data::satelliteはnamespace)

これは実装ファイルであるData.cppで宣言されています。

一方でData.hには、例えば(object::data::)satellite::Gpuのようなクラスが用意され、各々これらにはアクセスを許すクラスを記述しています。

外部のclass用にヘッダを用意。gpu::GpuBaseクラスのみがsatellite::Gpuにアクセスできます。 (gpuはnamespace)

satellite::Gpuの関数はData.cppに定義され、DataBaseクラスの関数を用いて記述されます

DataBaseは実装ファイルに宣言するため、これにアクセスする関数は必ず同ファイルに定義されます。DataBaseを用いる処理は一所に集約され、管理しやすいとも言えると思います。

(Data.h)

```
class object::data::satellite::Gpu {

    friend class gpu::GpuBase;

    static const ARRAY<float>&
        LoadVertex(const std::string& vertexfile);
};
```

(ARRAYは自作のコンテナクラス)

(Data.cpp)

```
class object::data::DataBase{

 public:
    static ARRAY<float> vertexbuff;
    (...)
};
using namespace object::data;

const ARRAY<float>& satellite::Gpu::
        LoadVertex(const std::string& vertexfile){

      (DataBase::vertexbuff等を用いて関数を記述)
}
```

関数定義

satellite::Gpuと同様な、異なるfriendクラスとstaticメンバを持つクラスを必要に応じて用意し、右のように(継承を用いて)1つのクラス(Data)にまとめています。

例えばgpu::GpuBaseクラスは

(object::data::)Data::LoadVertex(filename);

のように、object::ObjectBaseクラスでは

(data::)Data::LoadDataText(引数);

のようにしてDataの関数にアクセスします。

疑似的ではあるかもしれませんが、1つの共通ヘッダを各classはincludeし、同じobject::data::Dataクラスにアクセスしつつも各々に必要なメンバ関数のみが見えている構造となっています。

Dataクラスにアクセスする側にとって楽な仕様であると思います。

(Data.h)

```
class object::data::satellite::Gpu {
   friend class gpu::GpuBase;
   ( LoadVertex(…) などのメンバ関数)
};
```

```
class object::data::satellite::Object{
   friend class object::ObjectBase;
   ( LoadDataText(…)などのメンバ関数)
};
```

```
class object::data::satellite::selfy::
                Creations {
   friend class object::data::DataBase;
   (メンバ関数)
};
```

(…)

```
class object::data::Data :
   public satellite::Gpu,
   public satellite::Object,
   public satellite::selfy::Creations,
   (…)
{};
```

[(番外) 同系列の実装ファイルが1枚のヘッダを共有する構造]

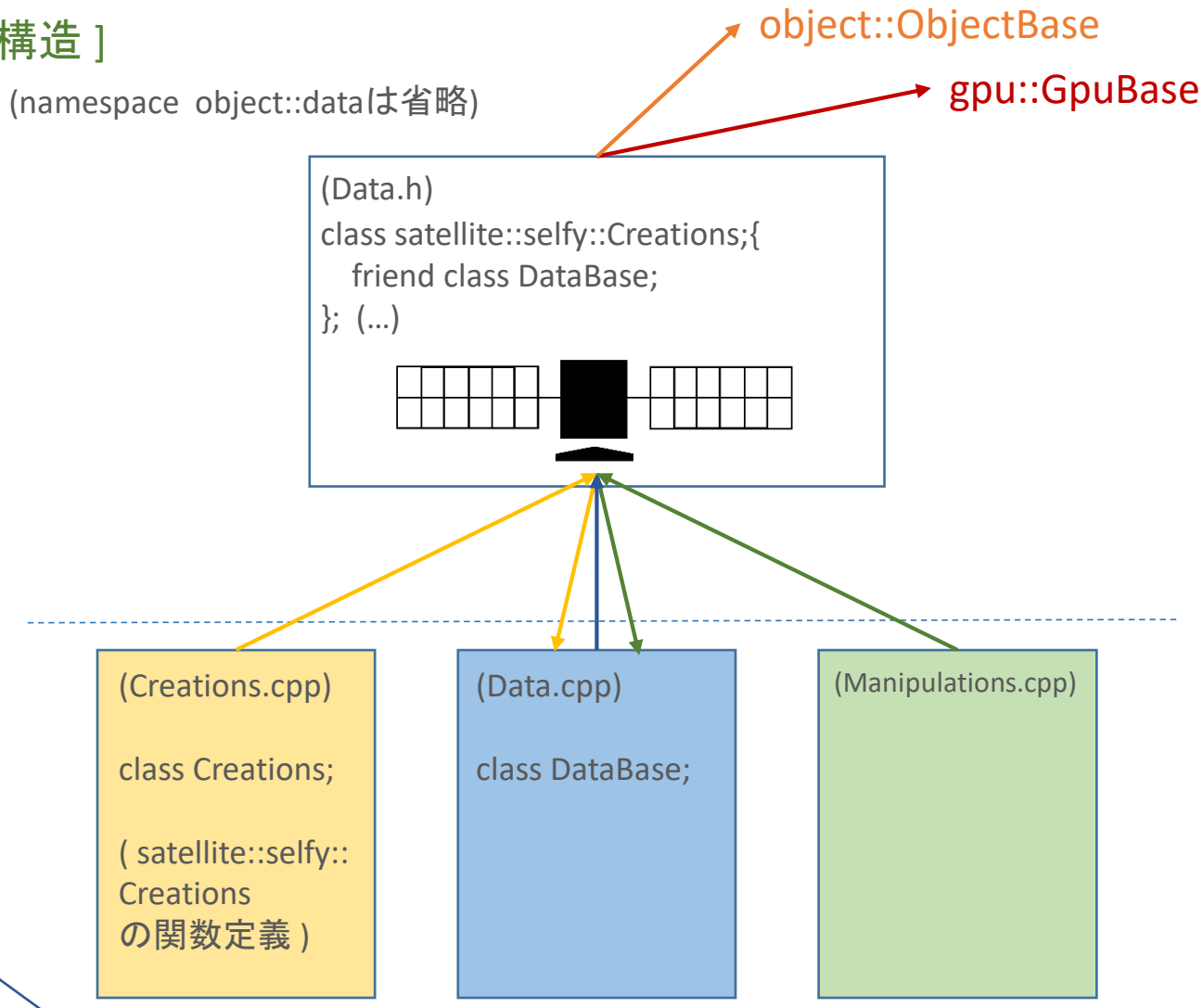
My Game Programming in C++では、Data.hとData.cpp
の2つ以外にもCreations.cppとManipulations.cpp
を”data”の同系列として構成しています。

2つは順に、頂点データの動的生成と(いわゆる)アクターの操作を記述した関数・クラスを並べたファイルであり、DataBaseクラス(Data.cpp)では名前を指定されるとこれらのファイルから該当関数・クラスを選びデータ・アクター(の生成関数)を提供します。

実装ファイル3枚は1枚のヘッダファイルを共有しています。Dataクラス(Data.h)は外部の系統のクラスからのアクセスを管理しているだけではなく、ここを介してData.cppから同系統のCreations.cpp等にアクセスする構成としています。

ヘッダ1枚だとincludeすべきヘッダはどれか相手に分かりやすいかと思います。

My Game Programming in C++では全てのstaticなクラスは、後述の非staticなクラスも含めて、この構造を採用しています。



satellite::selfy::Creationsクラス（Data.h）にはDataBaseクラス(Data.cpp)がアクセスでき、メンバ関数の定義はCreations.cppに記述されています。このクラスを通してDataBaseはCreations.cppのstatic関数にアクセスします。

(コンパクトな構造？)

[friendを用いたアクセス制御 (非staticメンバの場合)]

非staticなメンバをもつクラスを作成する場合に、必ずしもstd::vectorのような汎用利用が想定される訳ではなく、中には特定のクラスのみにメンバへのアクセスを許したい場合があるかと思います。

My Game Programming in C++の(audio::)Audioクラス(Audio.h)は個々のアクターの持つサウンド関連のデータ(非staticなメンバ)を扱います。

InstanceManagerクラスはサウンド関連データをまとめたクラスの参照となります。

非staticメンバを持つクラスを作成する場合、これまでのようなstaticなクラスも1つのクラスに統合した上で、統合先(ここではaudio::Audio)に非staticメンバを記述しています。

(Audio.h)

```
class audio::satellite::Camera {
    friend class camera::CameraBase;
    (staticなメンバ)
};                                          (cameraはnamespace)
```

```
class audio::satellite::Game {
    friend class game::GameBase;
    (staticなメンバ)
};                                          (gameはnamespace)
```

```
class audio::Audio : virtual public object::ObjectCore,
    public satellite::Camera, public satellite::Game {

    friend class object::Object;  (objectはnamespace)
    friend class object::ObjectBase;

    InstanceManager& manager;

    Audio(void);     // コンストラクタ
    (その他の非staticなメンバ関数)
};
```

object::ObjectCoreに関しては本題と逸れるので説明は省略します

(audio::)Audioクラスでは、friend宣言を用いて関数だけでなくコンストラクタを含めた全メンバへのアクセスも制限しています。

一方でこのためfriendクラスはAudioクラス内の全てにアクセスできてしまいますが、そのfriendクラスのために用意したメンバ以外にはクラス宣言が不明の参照型の変数(manager)しかありません。

Audioクラスはpimplイディオムとなります。参照にしたことでInstanceManagerクラスの内訳の変更に対して、Audio.hをincludeしているファイルの再コンパイルが必要ないという利点もあります。

manager はObject/ObjectBaseクラスに露出していますが、宣言が不明のため何もできないことと、隠すための細工の手間を考えて、これでよしとしました。

staticなクラスの場合と同様に、サウンドのデータ処理に関連するクラスは全て実装ファイル内で宣言することで一所で全処理を把握できるようにしています。

(Audio.h)

```
class audio::Audio : virtual public object::ObjectCore,
    public satellite::Camera, public satellite::Game {

    friend class object::Object;  (objectはnamespace)
    friend class object::ObjectBase;

    InstanceManager& manager;

    Audio(void);    // コンストラクタ
    (その他の非staticなメンバ関数)
};
```

(Audio.cpp)

```
class audio::AudioBase { (…) };
class audio::InstanceInfo { (…) };
class audio::InstanceManager { (…) };
(…)
```

関数定義

勿論object::Object/ObjectBaseクラスはsatellite::Camera, satellite::Gameクラスのメンバにはアクセスできません

[protectedを用いつつ特定クラスだけにアクセスを許したい場合]

My Game Programming in C++ のobject::Objectクラスではアクターの基本操作を行う関数が用意されています。

Objectクラスを継承した派生クラスでそれらを組み合わせて関数FirstActとSecondActを記述し、アクターの挙動を取り決めます。

このためどうしてもprotectedを用いることになりますが、想定していないクラスからのアクセスが全くないとは言い切れなくなるかと思います。

どこかで勝手に派生クラスを作成して走らせている、のような話ですが。

(Object.h)

```
class object::Object :
public satellite::Camera, public satellite::GameScript,
public satellite::Game {

 protected:
   ( アクターの基本操作を行う関数 )
   Object(ObjectBase& objbase);    // コンストラクタ

 public:
   ObjectBase& objbase;

   virtual void FirstAct(void) = 0;
   virtual void SecondAct(void) = 0;
  (...)

};
```

そこでMy Game Programming in C++では、実装ファイル(Object.cpp)で宣言しているObjectBaseクラスへの参照をObjectクラスに持たせ、アクターの基本操作関数はこの参照を介してアクターの情報にアクセスできるようにしています。

ObjectBaseクラスは先程のAudioクラスの他にGpuクラス(アクターのグラフィック関連のデータを扱う)、Actionクラス(アクターの衝突関連のデータを扱う)を統合したクラスになります。つまりObjectBaseクラスは各アクター固有の情報を保持したクラスになります。

先程のAudioクラスと違うのは、Objectクラスのコンストラクタから参照を渡さなければならないことで、(ObjectBaseクラスは実装ファイルで宣言されているために)仮に予期しない場所でオブジェクトを生成されたとしても有効な参照先を持たず、アクターの情報にアクセスできません。

ObjectBaseクラスを生成できるのはObject.cpp内のみであり、よってここでのみObjectクラスを有効なものにできます。このファイルの外で勝手にObjectが使われることを防ぎ、Objectクラスの管理を楽にします。

(Object.h)

```
class object::Object : (…) {
 protected:
   Object(ObjectBase& objbase);    // コンストラクタ
  (…)
 public:
   ObjectBase& objbase;
  (…)
};
```

(Object.cpp)

```
class object::ObjectBase : virtual public object::ObjectCore,
public gpu::Gpu, public audio::Audio, public action::Action
{  (staticメンバ)
   ObjectBase(引数);   // コンストラクタ
   (非staticメンバ)
 };
using namespace object;
ObjectBase::ObjectBase(引数) {
//ここで自身のポインタを渡してObject(の派生クラス)を動的に生成
}
Object::(アクターの基本操作関数){
 // objbase.(関数)を用いてアクターのデータにアクセス
}
```

[まとめ]

- 相手クラスごとにアクセスを制限したクラスを介して外部クラスを自クラスにアクセスさせることで、意図しないクラスからのアクセスを防ぎつつ、アクセスルートをはっきりさせることで管理を容易にする。

- アクセスを制限したクラスを右のように1つのクラスにまとめることで、統一されたクラス名からアクセスできるようにし使い勝手をよくする。

- staticなクラスを実装ファイルに宣言することでクラスへのアクセスを同ファイル内のみからに制限し、管理しやすくする。

- 1枚のヘッダを介して同系列のファイルまたは外部と交流すれば、コンパクトな構造になり外部にとっても分かりやすくできるか。

- 非staticメンバはpimplイディオムを用いるなどしてデータに関わるクラスを実装ファイルに宣言させ、データの管理を容易にし、仕様変更にも強くする。宣言不明の参照が露出することは気にしない。

- protectedを用いる場合は部外者には生成できない参照(またはポインタ)を必要とさせることで使用を制限する。

```
class Audio {
    GameBase:
        static void funcs(void);
    CameraBase:
        static void funcs(void);
    Object / ObjectBase:
        void funcs(void);
};
```

同系列のファイルは同じ人・チームが管理していて、チームで分担してソースコードを作成・まとめあげる場合にこれらの方法は活きてくるのではないかと思います。

ありがとうございました
Thank You !

Atsuhito Fukuda