ゼルダから学ぶメモリ管理

ノーマル（最初から）

メモリ管理の
必要性とその手法

ハード（応用）

時のオカリナから学ぶ
メモリ管理

メモリ管理の必要性とその手法 1

まず初めに、「なぜメモリ管理が必要なのか？」ということを解説していきます。

メモリ管理する、主な理由としては、

「上限のあるメモリ容量を、より効率良く使うため」というものがあります。

任天堂では、

プランナーやデザイナーであっても、メモリマップについて学ぶそうです。

だからという理由ではありませんが、知っておいて損はないと思います。

メモリ管理の必要性とその手法２

そこでまず、

メモリ管理する上で、発生する問題やその解決策について、先に解説していきます。

メモリ管理をする上で発生する問題は主に、

「メモリリーク」「メモリの二重解放」「メモリの断片化（フラグメンテーション）」が挙げられます。

メモリ管理の必要性とその手法３

#メモリリーク

```
13  int main()
14  {
15      Object* object = new Object();
16      InitID(object);
17  }
```

#二重解放

```
30  _CRT_SECURITYCRITICAL_ATTRIBUTE
31  void __CRTDECL operator delete(void* const block) noexcept
32  {
33      #ifdef _DEBUG
34      _free_dbg(block, _UNKNOWN_BLOCK);
35      #else
36      free(block);
37      #endif
38  }
39
```

例外がスローされました

0x00007FFB9B76024E (ucrtbased.dll) で例外がスローされました (SceneManager.exe 内): 0xC0000005: 場所 0x000000000000810F の読み取り中にアクセス違反が発生しました

詳細のコピー

▲ 例外設定

☑ この例外の種類がスローされたときに中断

次からスローされた場合を除く:

☐ ucrtbased.dll

例外設定を開く | 条件の編集

まずは、「メモリリーク」と「メモリの二重解放」について、

メモリリークとは、確保したメモリを、
解放せずに放置してしまうことを言います。

二重解放とは、一度解放した（未割り当ても）メモリを、
再び解放処理してしまうと発生します。

メモリ管理の必要性とその手法 4

#イメージ

```
14  int main()
15  {
16      // 関数を抜ける時、自動で解放してくれる。
17      std::unique_ptr<Object> object;
18      object = std::make_unique<Object>();
19  }
```

先ほどの二つの問題は、

C++言語では、比較的簡単に解決することが出来ます。

それが、「スマートポインタ」です。

左のコードでは「std::unique_ptr」という部分ですね。

基本的に、皆さんが作る作品は、これだけで十分です。

メモリ管理の必要性とその手法５

#GC

```
0 個の参照
void TestMethod()
{
    // Unity C# は確保すると、
    var test = new Test();

    // 関数終了時に、自動で解放してくれる
}
```

#イメージ

```
0 個の参照
void TestMethod()
{
    // UnityC# には GC を停止させる機能もある。
    GarbageCollector.GCMode = GarbageCollector.Mode.Manual;

    // その際は、明示的にGCを実行しなければいけない。
    GarbageCollector.CollectIncremental();
}
```

処理負荷が高い場所では、予めGCを切っておく等、工夫が考えれる

また「メモリリーク」と「メモリの二重解放」を解決するには、
もう一つ、「ガベージコレクション」というものもあります。

C#や、Javaなどの言語は、元々搭載されている機能です。

しかし、プログラムが停止されてしまうなど、
デメリットも存在するので、使用する際は工夫も必要です。

メモリ管理の必要性とその手法６

基本的な作品では、先ほどまでに紹介したもので、十分こと足りますが、
AAA作品や低スペックな環境時などは、より細かい問題が残っています。

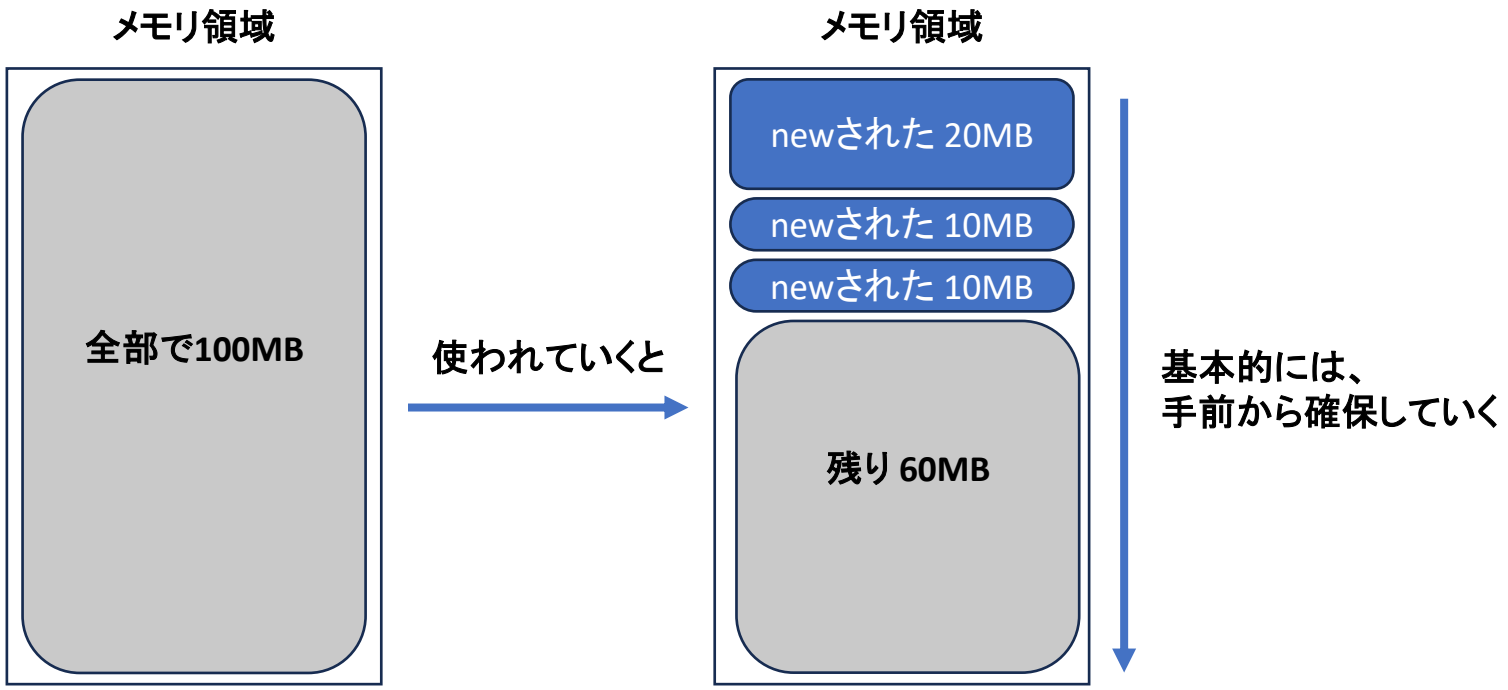
それが「メモリの断片化（フラグメンテーション）」という問題です。

メモリの断片化は、先ほどまでの分かりやすい問題と違い、
中々感じにくい問題なので、ここから順を追って説明していきます。

メモリ管理の必要性とその手法 7

まずは、メモリの確保の流れを考えていきましょう。

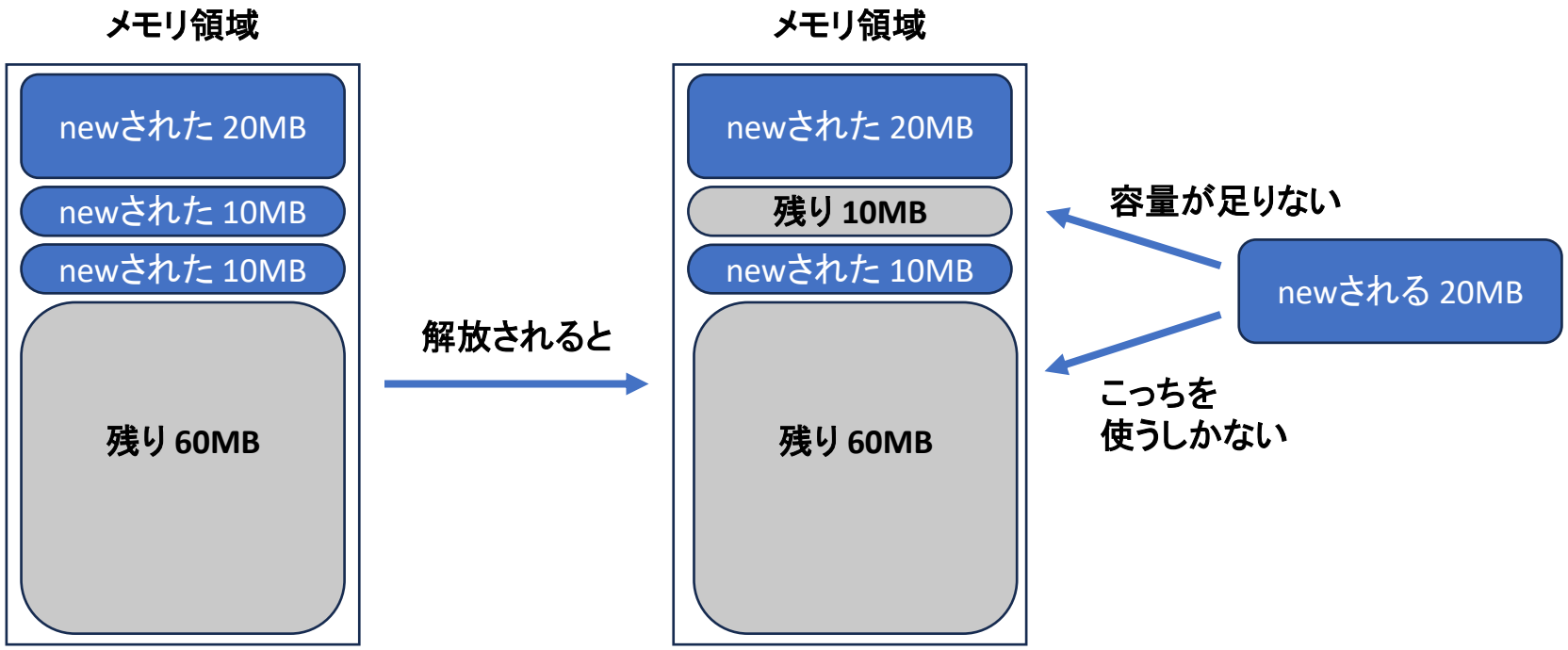
メモリを確保する際、基本的には、下記のような形になります。



メモリ管理の必要性とその手法８

基本的にnewされた順番で解放されれば問題ありませんが、通常はそうはいきません。

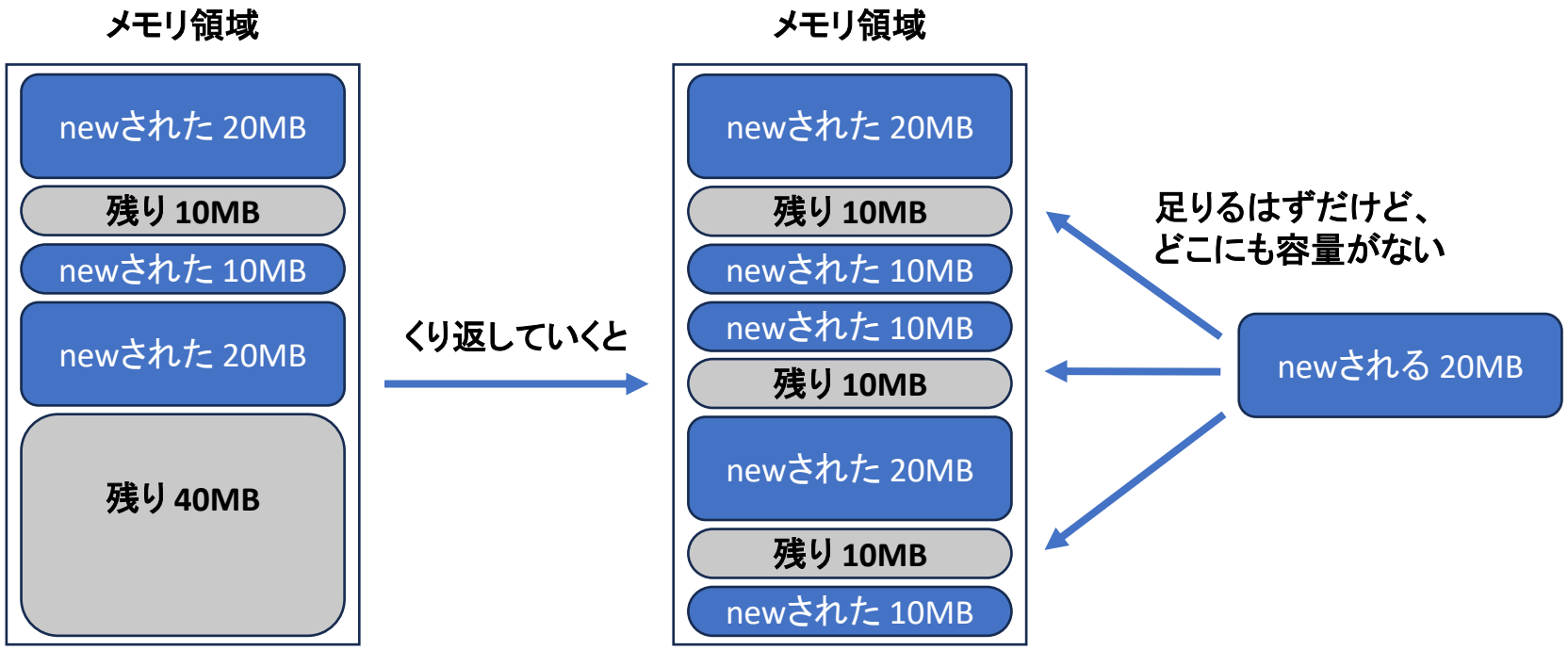
下記のような場合では、次のnew要求の際、空きメモリを飛ばして確保していきます。



メモリ管理の必要性とその手法９

先ほどのように、空き容量があるにも関わらず、

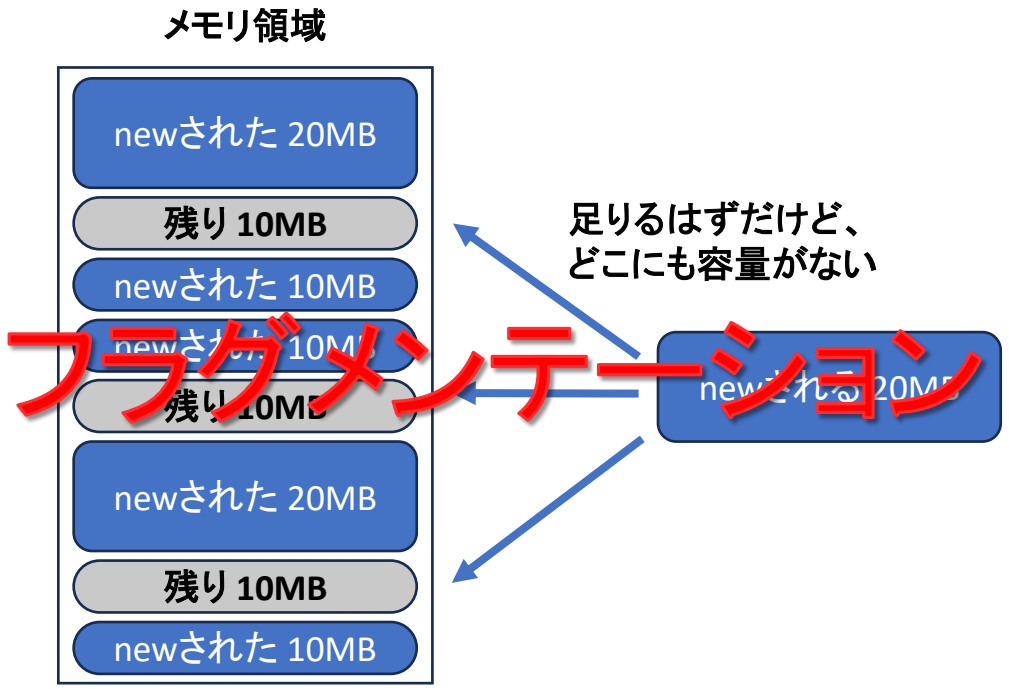
容量が足りず、次のメモリを確保することをくり返していくと、下記のようになるはずです。



メモリ管理の必要性とその手法 10

そしてこのように、

連続した未使用領域が少なくなる状況を「メモリの断片化（フラグメンテーション）」と言います。



メモリ管理の必要性とその手法 11

メモリが断片化すると、

新たなメモリ確保や、走査に時間がかかるのでパフォーマンスが落ちてしまいます。

また、断片化する主な要因としては、

長時間のアプリケーションの使用、または小さいメモリを沢山使うなどの理由があります。

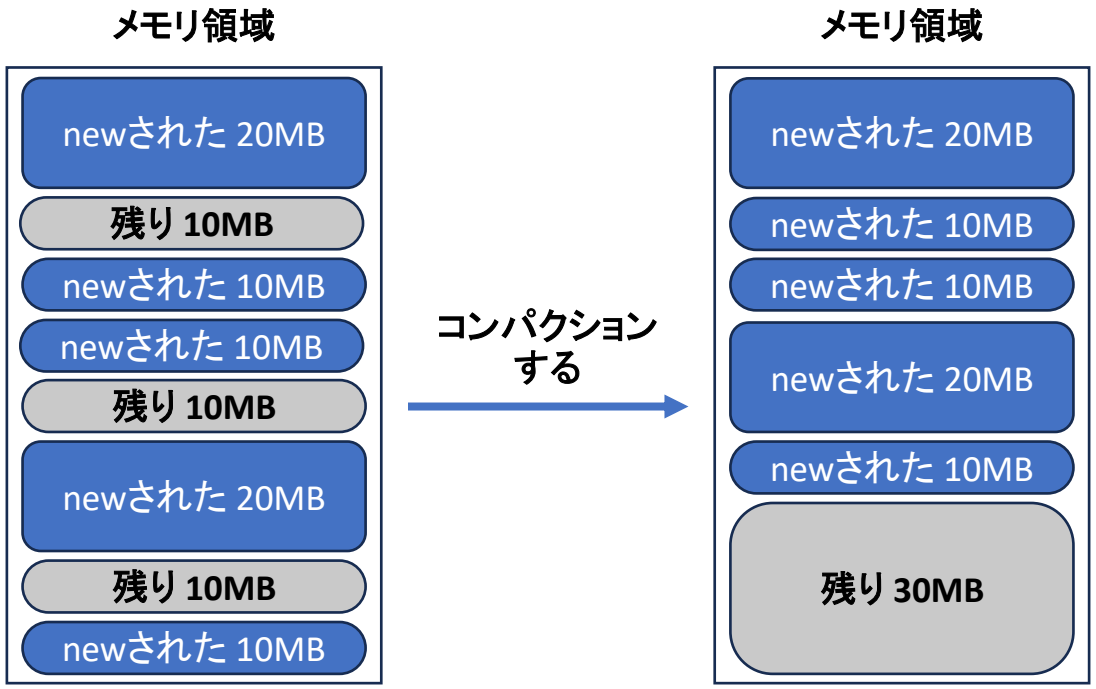
そして、断片化を解消する方法としては、

「メモリコンパクション（デフラグメンテーション）」という方法があります。

メモリ管理の必要性とその手法 12

メモリコンパクション（デフラグメンテーション）とは、

バラバラに配置された空き容量を、再配置することで、一つにまとめようとするものです。



メモリ管理の必要性とその手法 13

メモリコンパクションは、GCとセットで搭載されていることがあります。

※「コピーGC」という名前です

そんな便利なメモリコンパクションですが、

実はGC同様に、プログラムを停止させるなどのデメリットも存在します。

そのため、「そもそもメモリの断片化が起こらないようにする」という方法も存在します。

次はそちらを解説していきます。

メモリ管理の必要性とその手法 14

そもそも、メモリの断片化を怒らないようにするにはどうすればよいか？

それは、「メモリの確保時に断片化が起こらないようにすること」です。
とてもシンプルですよね。

「メモリアロケーター」というシステムを使い、メモリの確保時に細工を施すことで実現させます。

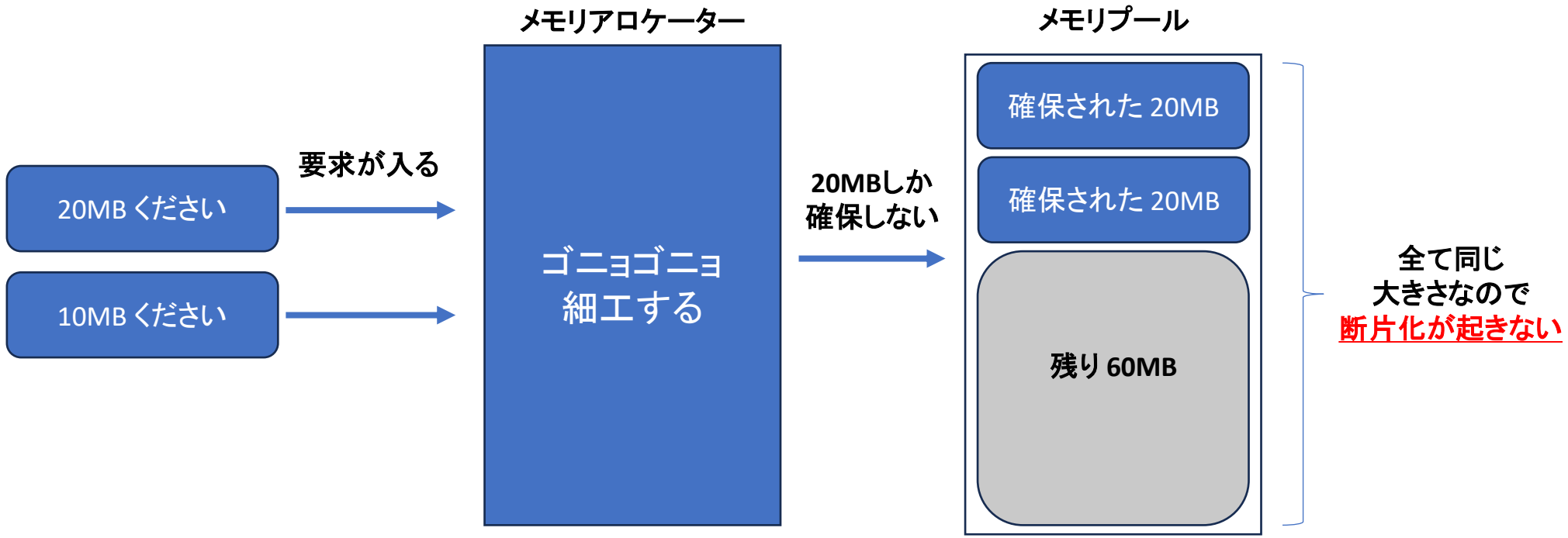
また、メモリアロケーターには沢山種類があるのですが、今回は
断片化を防ぐことが出来る、「メモリプールアロケーター」というものを紹介、解説していきます。

メモリ管理の必要性とその手法 15

メモリプールアロケーターは、

予め用意した固定長サイズの領域から、固定サイズのメモリしか確保しません。

※下記は極端な例

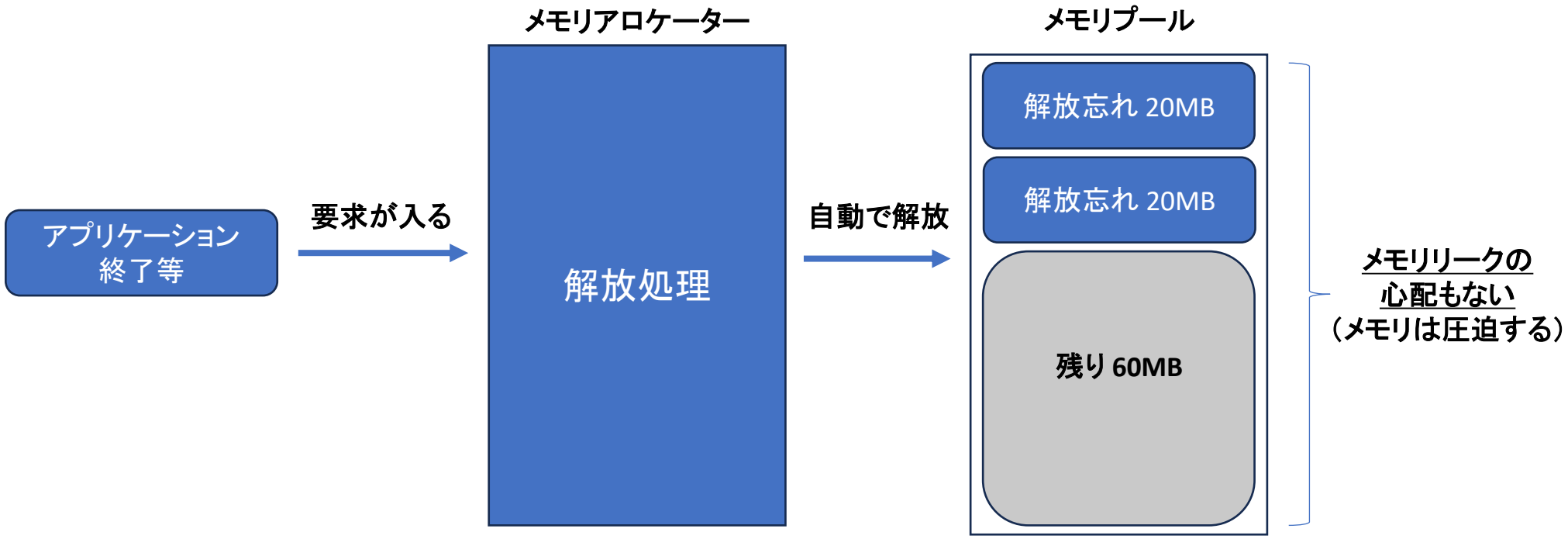


メモリ管理の必要性とその手法 16

また、確保するメモリサイズが同じため、

メモリの確保、解放に必要なの処理時間が常に一定になるなどメリットが多いです。

※メモリプールはめちゃくちゃ優秀

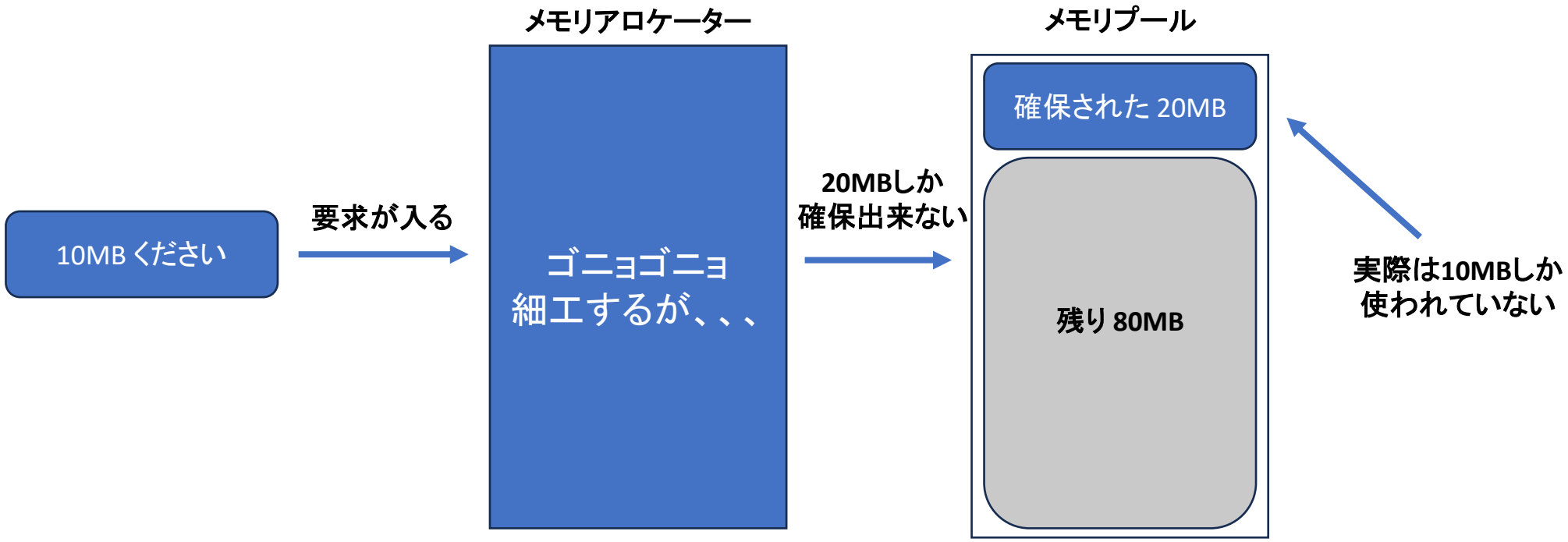


メモリ管理の必要性とその手法 17

勿論、メモリプールにも欠点はあります。

それは、メモリサイズを固定にするため、無駄なメモリを消費してしまうことです。

※工夫次第で全然良くなりますし、手軽に使用出来るので、使わないのは勿体ないです。



メモリ管理の必要性とその手法 18

メモリプールアロケーターを実装する際は、

大体16バイト～256バイトまでの5つのプールを用意しておき、

指定された容量に合わせてメモリ割り当てをするようにすると良いと思います。

ですが、C++には

「memory_resource」というのがあるので、自分で実装することはないと思います。

メモリ管理の必要性とその手法 まとめ

ここまで、

メモリ管理の必要性と、メモリ管理する上での問題とその解決についていくつか紹介しました。

正直、「これ学んでどうするの？」と、皆さん同じことを思うかもしれませんが、

このメモリ管理の積み重ねの先に、ゼルダ等の素晴らしい作品達があります。

メモリ管理は、一人だけ意識したところで、どうにかなるものではありません。

チームメンバー全員がメモリについて知識を持ち、作品を作っていく他ありません。

ですので、記憶の片隅にでも覚えておいて頂ければ幸いです。

ゼルダから学ぶメモリ管理（応用）

1. 時のオカリナでのメモリ管理事例

ゼルダから学ぶメモリ管理（時のオカリナ編）１

ここからは「ゼルダ 時のオカリナ」から、メモリ管理について、更に深堀りしていきます。

「めっちゃ古い作品じゃない？」って思われるかもしれませんが、
メモリ管理初心者としては、非常に学ぶことも多いので良いサンプルだと思います。

ここからは、内部処理についても解説しますが、ちゃんと裏どりはしています。
（世の中にはRATのために、ROM抜きをして解読する人がいるのです、、、すごい）

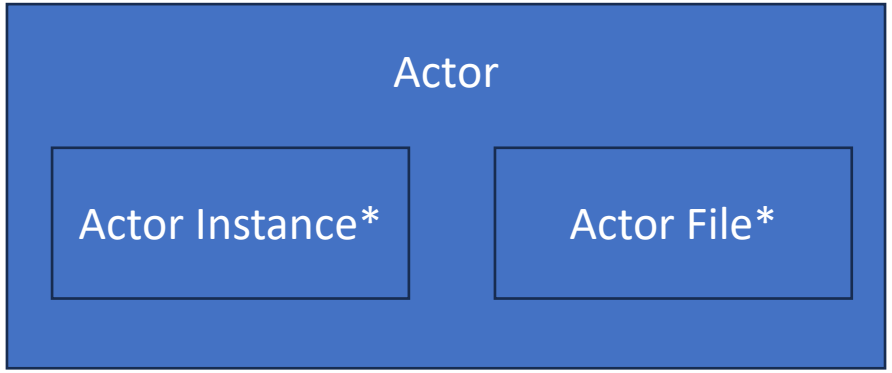
ゼルダから学ぶメモリ管理（時のオカリナ編）２

「ゼルダ　時のオカリナ」では、

一つのオブジェクトのことを「Actor」という単位で管理しています。

そして「Actor」は、

「Actor Instance」と「Actor File」の二つの情報で構成されています。



ゼルダから学ぶメモリ管理（時のオカリナ編）３

「Actor Instance」は、

生成されたアクターの座標、向きなどの情報保持しています。

「Actor File」は、

「何のアイテムなのか？」「どういう挙動をするのか？」「宝箱の中身は何か？」など、

生成されたアクターの情報を保持しています。

また「Actor File」は、キャラクターの種類ごとで一つのデータを共有しています。

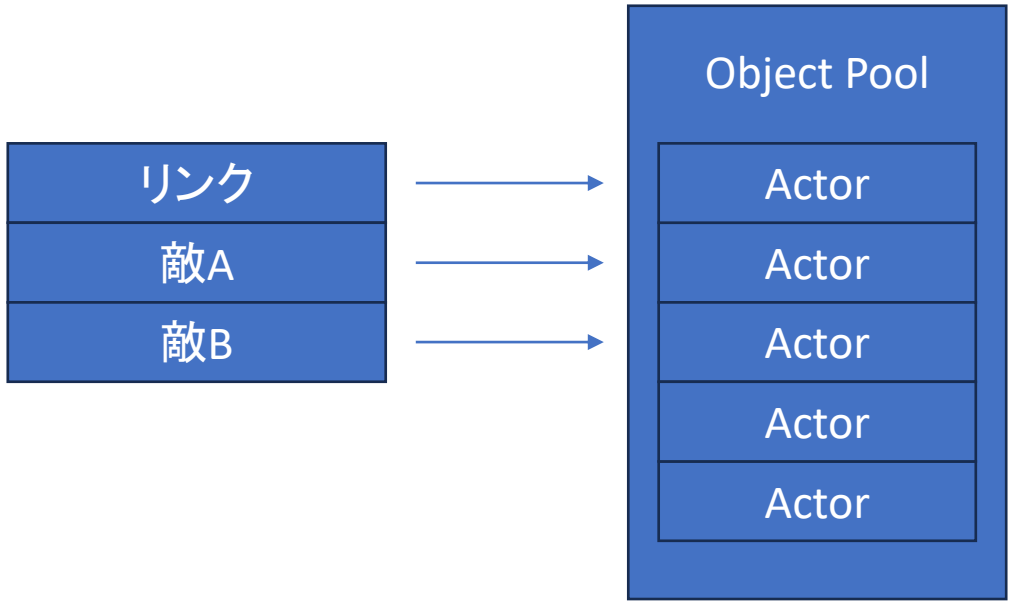
ゼルダから学ぶメモリ管理（時のオカリナ編）４

ゲーム内のキャラクターはメモリを共有しているというので、

オブジェクトプールで管理されているというのが分かります。

※これを利用したRTAバグがあるので正しいと思います。

イメージ

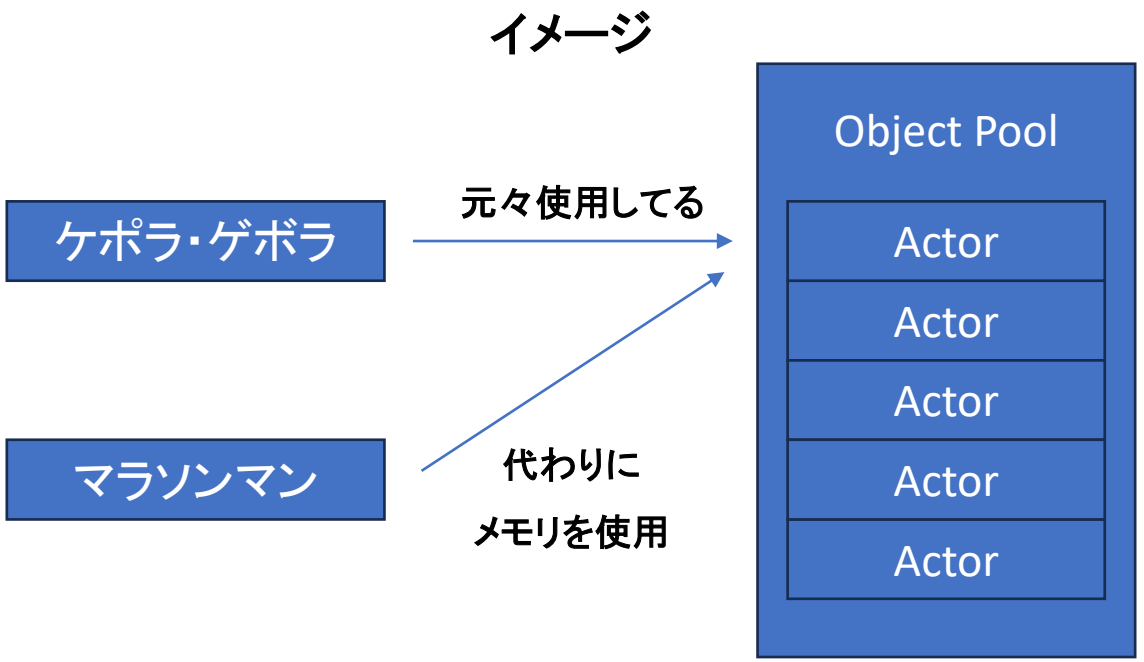


ゼルダから学ぶメモリ管理（時のオカリナ編）５

具体例としては、

ハイラル平原では<u>ケポラ・ゲボラという鳥が登場するのですが、</u>

<u>イベントが終わると解放されて、そこにマラソンマンが入りメモリを使用する</u>という感じです。



ゼルダから学ぶメモリ管理（時のオカリナ編）６

オブジェクトプールで管理する理由は、

ハードの性能から、表示できるキャラクター数の上限が既に分かっていたためだと思います。

そして「Actor Instance」と「Actor File」を分離することで、

Actor自体の総メモリサイズ（ポインタ二つと各データサイズ）が分かっているため、

ゲーム全体の総使用メモリサイズをプログラマー側が制御できる形にしているのだと思われます。

※ニンテンドー６４は64bitで制御可能ですが、32bitなはずなのでポインタサイズは4バイトになります。（32bitの理由は割愛します）

Actor全体の総メモリ使用量は、

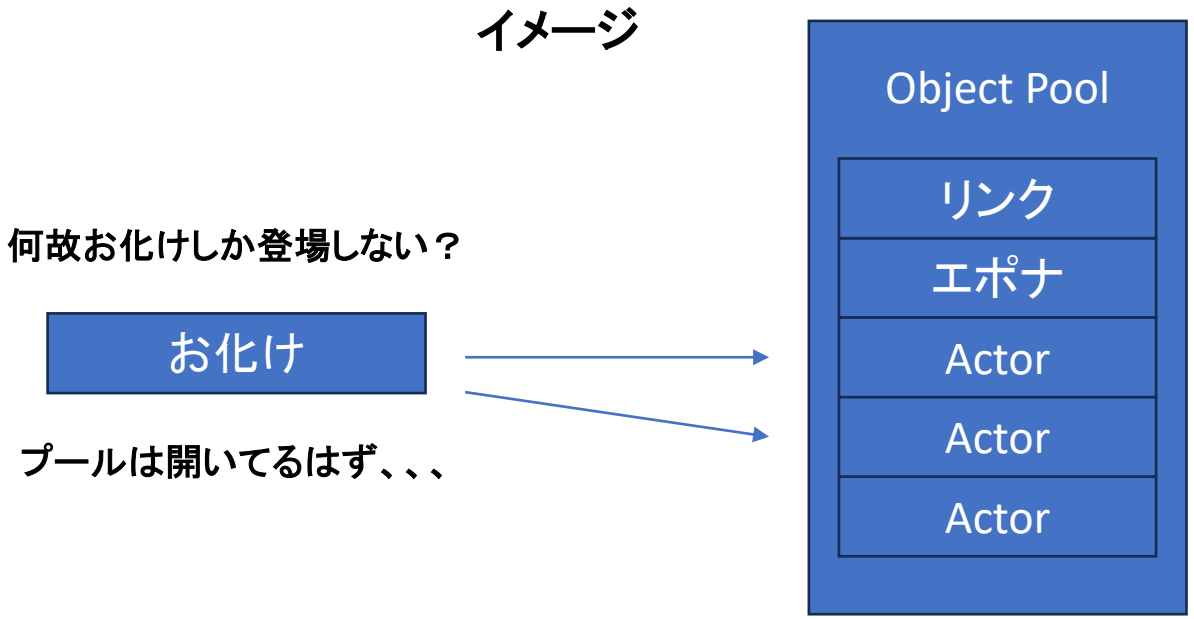
（（Actor Instanceサイズ ＋ 8バイト）× キャラクター数）＋（Actor Fileサイズ × キャラクタータイプ数）

ゼルダから学ぶメモリ管理（時のオカリナ編）7

ここから更に深掘りします。

リンクが大人になると、愛馬のエポナが登場するのですが、

そうすると、ハイラル平原では、敵キャラクターはお化け（ポウ）しか登場しなくなります。



ゼルダから学ぶメモリ管理（時のオカリナ編）８

ハイラル平原で登場するキャラクター一覧（この図から面白いことがわかります）

共通

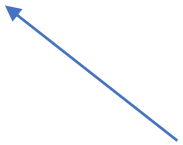
青テクライト、アキンドナッツ、大スタルチュラ

子供時代

ピーハット（オス・メス）、幼生ピーハット、スタルベビー（夜限定）、ケポラ・ゲボラ、マラソンマン

大人時代

エポナ、ポウ、ビックポウ



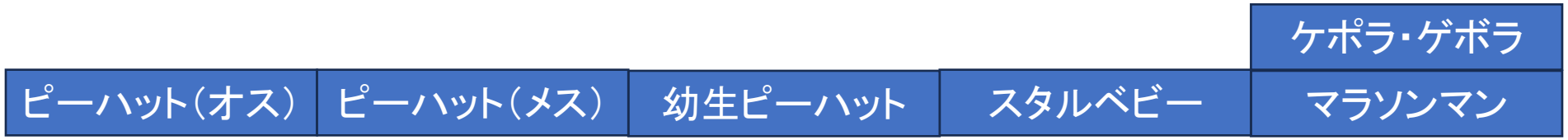
子供時代が以上にキャラクターが多い？何故？

ゼルダから学ぶメモリ管理（時のオカリナ編）９

ケポラ・ゲボラとマラソンマンはメモリを共有しているため、一つと考えることが出来ます。

そのため、バランスは下記の様になるはずです。（まだ全然、数が合わない、、、）

子供時代



大人時代

ゼルダから学ぶメモリ管理（時のオカリナ編）１０

先ほどの図で大体のバランスはイメージ出来ましたが、

子供時代と大人時代では、全然登場キャラクター数が合いません。

「それは何故でしょう？」

もちろん、カラクリが仕掛けられています。ミソはピーハットです。

ゼルダから学ぶメモリ管理（時のオカリナ編）１１

ピーハットは、オス、メス、そして子供の３種類いますが、

メモリ使用量を見ると、カラクリがあります。

それは、

ピーハットのオス・メスでは、テクスチャとモデルを共有しています。

ピーハットの子供は、テクスチャを共有して動作しています。

そこから各キャラクターのFile情報と、アニメーションデータを考慮すると、次の様になります。

ゼルダから学ぶメモリ管理（時のオカリナ編）１２

子供時代

他の１．５倍のサイズになる

大人時代

実は、こっちも他の１．５倍のサイズ

エポナは、他のキャラクターに比べ、モーション数や処理が多いため１．５倍程になる。

ゼルダから学ぶメモリ管理（時のオカリナ編）１３

やっとカラクリが分かりましたね。

ピーハットは、エポナと同じメモリ使用量で、
バリエーションが出るキャラクターとして考案された敵でした。

これをゲームデザイナーが考慮して、
仕様として出していることが衝撃ですが、宮本さんなのでね。

ゼルダから学ぶメモリ管理（時のオカリナ編）まとめ

ゼルダ 時のオカリナより、

メモリ管理について色々学んできたのですが、いかがだったでしょうか？

オブジェクトプールを使用した断片化防止と、リソースも考慮した総メモリの管理など、

多くのメモリ管理の工夫がされていたと思います。

当時ほどメモリは枯渇していませんが、まだメモリが無限にあるわけではありません。

AAAタイトルなどはメモリ管理との戦いですので、これを機にメモリ管理について知ってもらえれば幸いです。